SIAM J. COMPUT.
Vol. 21, No. 1, pp. 1-23, February 1992
© 1992 Society for Industrial and Applied Mathematics
001

EFFICIENT MOTION PLANNING FOR AN *L*-SHAPED OBJECT*

DAN HALPERIN†, MARK H. OVERMARS‡, AND MICHA SHARIR†§

Abstract. An algorithm that solves the following motion-planning problem is presented. Given an L-shaped body L and a two-dimensional region with n point obstacles, decide whether there is a continuous motion connecting two given positions and orientations of L during which L avoids collision with the obstacles. The algorithm requires $O(n^2 \log^2 n)$ time and $O(n^2)$ storage. The algorithm is a variant of the cell-decomposition technique of the configuration space [D. Leven and M. Sharir, *J. Algorithms*, 8 (1987), pp. 192–215], [J. T. Schwartz and M. Sharir, *Comm. Pure Appl. Math.*, 36 (1983), pp. 345–398], but it employs a new and efficient technique for obtaining a compact representation of the free space, which results in a saving of nearly an order of magnitude. The approach used in our algorithm is also applicable to motion planning of certain robotic arms whose spaces of free placements have a structure similar to that of the L-shaped body.

Key words. computational geometry, robotics, motion planning, data structures, arrangements, configuration space, dynamic segment trees

AMS(MOS) subject classifications. 68U05, 68Q25

1. Introduction. Let B be a robot system having k degrees of freedom that is free to move within a two- or three-dimensional domain V which is bounded by various obstacles whose geometry is known to the system. The motion-planning problem for B is, given the initial and desired final position of the system B, to determine whether there exists a continuous motion from the initial position to the final one, during which B avoids collision with the known obstacles, and if so, to plan such a motion.

Since the general motion-planning problem is very hard, a significant effort was devoted to develop efficient algorithms for some special cases, particularly that of motion planning for rigid objects in a two-dimensional polygonal space. To get some feeling of what "efficient" means in this context, we note that such moving systems B have three degrees of freedom, so their *configuration space*, i.e., the space of parametric representations of placements of B, is three-dimensional. Let n denote the number of obstacle corners, and suppose that the complexity of B is constant. Then the "free" portion FP of the configuration space, consisting of placements of B in which it does not meet any obstacle, is bounded by $O(n)$ (algebraic) *collision-constraint* surfaces, each being the locus of placements where some specific feature of B makes contact with some specific obstacle feature. By standard arguments from algebraic geometry, the complexity of FP is $O(n^3)$. Moreover, the recent general technique of Canny [Ca] yields a (fairly complicated) algorithm that computes a discrete representation of FP in time $O(n^3 \log n)$. A more specialized algorithm has recently been obtained by Avnaim, Boissonnat, and Faverjon [ABF], whose complexity is also $O(n^3 \log n)$.

Thus the general goal of studying motion-planning problems for rigid objects in the plane is to obtain subcubic, and ideally near-quadratic, algorithms. Ke and O'Rourke [KeO] give a quadratic lower bound for the actual combinatorial complexity

* Received by the editors July 5, 1989; accepted for publication (in revised form) September 4, 1990. The work of the first and third authors was supported in part by Office of Naval Research grant N00014-90-J-1284, National Science Foundation grant CCR-89-01484, and grants from the U.S.-Israeli Binational Science Foundation, the Fund for Basic Research of the Israeli Academy of Sciences, and the German-Israeli Foundation for Scientific Research and Development.

† School of Mathematical Sciences, Tel-Aviv University, Tel-Aviv 69978, Israel.

‡ Department of Computer Science, University of Utrecht, Utrecht, the Netherlands.

§ Courant Institute of Mathematical Sciences, New York University, New York, New York 10012.

of a collision-free motion for a line segment (see [KeO] for a definition of motion complexity). The lower bound for the decision problem, that is, to decide whether a motion exists, is not known to be quadratic. However, the only cases where near-quadratic algorithms have been obtained involve *convex* objects (see [LS], [SiS] for the case of a line segment and [CK], [KS2] for the case of an arbitrary convex polygon). Efficient motion-planning algorithms for nonconvex polygons in two-dimensional polygonal space have scarcely been dealt with. An early consideration of the problem appears in Schwartz and Sharir [SS], where they extend their $O(n^5)$ projection algorithm for moving a line segment to a nonconvex polygon B with similar running time, assuming that B has a fixed number of vertices. As just noted, the algorithms of [Ca] and [ABF] already improve the complexity to $O(n^3 \log n)$, but no better solutions were known in the nonconvex case.

In this paper we present a new approach to motion planning of nonconvex objects in the plane. We obtain a near-quadratic algorithm for solving this problem in the special case of an L-shaped object moving amidst a collection of point obstacles in the plane.

We distinguish between the *reachability problem*, which is to check whether a continuous collision-free path from the initial to the final placement exists, and the *find-path problem*, which is to actually compute such a path if it exists. The former is the concern of this paper, and the latter is discussed in an accompanying paper [HS].

Our algorithm uses the *decomposition* approach to motion planning [SS], [LS], [KS2], which partitions the space FP of free placements of the robot system into a finite number of simple, connected cells. These cells define vertices in a so-called *connectivity graph CG*. Two cells are adjacent in CG if they have a common boundary enabling a direct crossing of the moving object between them. It can be shown that in the worst case, the space FP for our L-robot has $\Omega(n^3)$ connected components, thus, in particular, its total combinatorial complexity can be $\Omega(n^3)$.

To see this, consider Fig. 1, where there are three sets of $n/3$ points each. Choose an interval between two successive points in the upper horizontal set; choose an interval between two successive points in the lower horizontal set. Now locate the "vertical" bar $\overline{qp}$ of L so that it will intersect the two chosen intervals. Finally, choose a pair of consecutive points in the vertical set and locate the "horizontal" bar $\overline{qr}$ of L such that it will also intersect the interval between this pair. It is easily verified that there are $\Omega(n^3)$ such choices and that L cannot continuously move between any pair of such

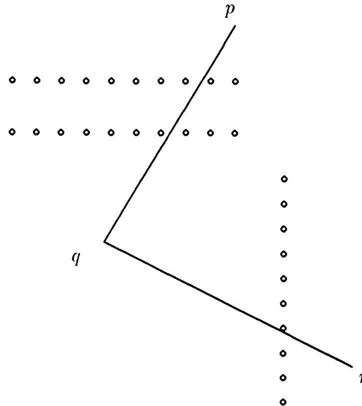


FIG. 1. *A configuration of obstacles with an $\Omega(n^3)$-size FP.*

placements, if the size of L and the location of the points are chosen in an appropriate manner.

However, using some interesting data structure techniques, we construct an implicit representation of FP using a compact connectivity graph that requires subcubic space and can be constructed in subcubic time. To be precise, our reachability algorithm requires $O(n^2 \log^2 n)$ time and $O(n^2)$ space.

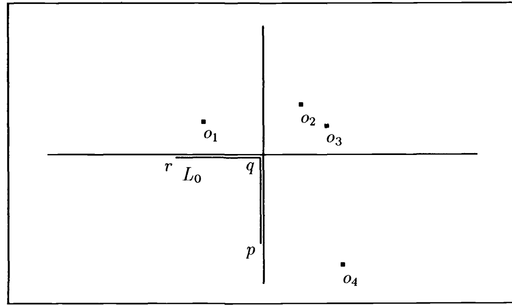
The paper is organized as follows. In § 2 we introduce the terminology, explore the basic ideas, derive some initial observations about the problem structure, and give an overview of our efficient solution. In § 3 we describe the data structures that support our algorithm. More details concerning the algorithm are described in § 4, where we also prove its correctness and analyze its complexity. We conclude in § 5 by discussing the novel ideas used in this paper and their potential applicability to other instances of the motion-planning problem, and by proposing some directions for further research.

2. Terminology and initial analysis. We denote the moving object L by pqr where $\overline{qp}$ is the vertical bar and $\overline{qr}$ is the horizontal bar of L (in some standard axis-parallel position). Thus p and r are the *external vertices* and q is the *internal vertex* of L. Each position of L can be specified as $Z=(X, \theta)$ where X is the position of q and θ is the orientation of $\overline{qp}$. For the purpose of our algorithm, we will always present X in a rotated coordinate frame in which θ becomes a downward vertical direction. We denote by AP the resulting three-dimensional space of all positions of L, which can be identified with $R^2 \times S^1$. The set of point-obstacles is denoted by $O=\{o_i \mid i=1, 2, \cdots, n\}$. For the sake of representation only, we will delimit the planar workspace V in which L is free to move by a sufficiently large rectangle, and assume that this rectangle rotates with the coordinate system, so that it always remains axis-parallel.

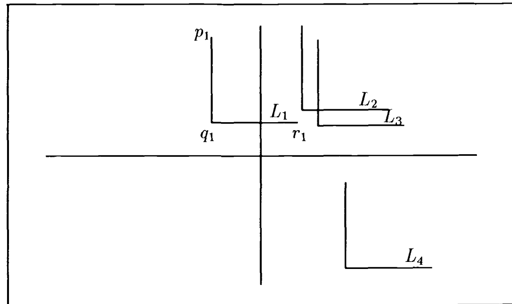
We shall call a position of L at which it does not touch any point a *free position.* The set FP of all free positions of L is an open three-dimensional submanifold of AP.

Similar to [LS] (see also [SS], [KS2]), our method decomposes FP into connected subcells of a simple form, using the following two-step approach. First we consider the case in which L is allowed to translate arbitrarily, but not to rotate. Then we consider the case in which L is allowed both to translate and to rotate in V.

To build a discrete representation of FP for a fixed orientation, we follow a common practice in motion planning [LW], [KS1], [KLPS], and compute the Minkowski (i.e., vector) difference of each obstacle and the robot. Let L_0 denote a position of L in a Cartesian coordinate system in which q coincides with the origin of the system, p lies on the negative y-axis, and r lies on the negative x-axis. For any fixed θ, and for $i=1, 2, \cdots, n$, let $L_i = L_i(\theta) = o_i(\theta) - L_0$ be the Minkowski difference of the obstacle o_i and L_0, in the rotated coordinate frame where θ points downwards; here $o_i(\theta)$ denotes the rotated position of the point obstacle o_i in this coordinate frame. The horizontal bar of L_i extends from $o_i(\theta)$ to the right, and the vertical bar extends from $o_i(\theta)$ upwards (see Fig. 2 for an illustration). Let $S_\theta = \{L_i(\theta) \mid i=1, 2, \cdots, n\}$ for some fixed orientation θ. S_θ defines a planar arrangement consisting of horizontal and vertical line segments. The cross-section of FP at a given θ, which we denote by FP_θ, is simply the complement of the union of the $L_i(\theta)$'s. In addition, to simplify the structure of S_θ, we add to it certain horizontal segments emanating from the vertices of the L_j's. We will refer to these extensions as *imaginary walls*; see Fig. 6(a) for an illustration (a formal definition is given below). S_θ, together with these extensions (and the big rectangle enclosing the workspace), divide FP_θ into orthogonal (axis-parallel) simply-connected polygons. We call each such polygon a *face* of FP_θ. Some of these faces are simply rectangles, in which case we call the four edges of the face the *northern*,



(a)



(b)

FIG. 2. *The expanded obstacles*: (a) L_0, $o_1 - o_4$; (b) $L_1 - L_4$.

southern, *western*, and *eastern walls* corresponding to the upper, lower, left, and right edges of that face. In other cases we call the rightmost edge of the face the *eastern wall.* The imaginary walls will be defined in such a way as to ensure that each face has a unique eastern wall.

If we let θ vary through the range $[0, 2\pi]$, the object $L_i(\theta)$ will trace a two-dimensional surface σ_i within AP; that is, $\sigma_i = \{(X, \theta) \mid \theta \in [0, 2\pi], X \in L_i(\theta)\}$. The collection of these surfaces forms an arrangement $\mathcal{A}$ of surfaces in AP, which decomposes the three-dimensional space AP into pairwise disjoint connected cells, each of zero, one, two, or three dimensions. We shall use the unquantified term *cell* for a three-dimensional cell of $\mathcal{A}$. A cell c of $\mathcal{A}$ is *interesting* if at least one cross-section of its closure $\bar{c}$ contains a vertex of some $L_i(\theta)$. All other cells of the arrangement are called *dull* (this terminology is borrowed from [AS]).

Remark 2.1. Formally speaking, the surfaces σ_i are not algebraic, because they depend trigonometrically on θ. However, they can be easily made algebraic if one replaces θ by, say, $t = \tan(\theta/2)$. For simplicity of exposition, we will continue to use θ as one of our coordinates, but consider this transformation as available whenever needed.

In our analysis, we will sometimes allow the motion of L to become *semifree*, namely, allow L to touch an obstacle. In the configuration space, the corresponding path is allowed to touch the union BFP of the surfaces σ_i. However, the path is not allowed to cross BFP transversally (which amounts in the physical space to L sweeping through an obstacle). We will therefore consider each σ_i to be "two-sided," where each side bounds, and can be reached from, a different portion of FP, and where no direct crossing from one side to the other is allowed.

How does the two-dimensional arrangement (of S_θ) change as the orientation θ of the original L varies? As the coordinate system rotates, FP_θ changes continuously, but its combinatorial structure remains unchanged, unless one of the following two types of critical events occurs at θ (see Fig. 3):

I. A vertex of one L_i meets an edge of another L_j.

II. Two parallel edges of two L_i's overlap.

These two types of events are substantially different in their effect on FP_θ. An event of type I has only a local effect on FP_θ—either a face is split into two, or two faces are merged into one. When an event of type II occurs, its effect is more global and causes four contiguous sets of faces, each set containing $O(n)$ (and, in the worst case, $\Omega(n)$) faces, to change some of their edges, to disappear, or to newly appear.

FIG. 3. *Events of type* I (*left*) *and type* II (*right*).

An orientation θ at which one of these events happens, will be called a *critical orientation.*

Remark 2.2. When two parallel edges overlap it is also the case that a vertex of one expanded obstacle meets an edge of another. We will treat this part of the type II event in a way similar to a type I event. Indeed, it will be more suitable to regard each overlap of two parallel edges as a combination of events of type I and of type II. From this point on, type II will refer only to the effect of the overlap on the "internal" faces involved, taking care of the "external" faces (i.e., the faces adjacent to the vertices of the overlapping edges) as effected by a type I event.

How many criticalities are there of each type?

LEMMA 2.1. *The number of critical orientations induced by a fixed pair of obstacles is bounded by a constant.*

Proof. The proof is trivial, by elementary geometric considerations. □

Since there are $\binom{n}{2}$ pairs of expanded obstacles, Corollary 2.2 follows.

COROLLARY 2.2. *There are $O(n^2)$ critical orientations of type* I *and $O(n^2)$ critical orientations of type* II; *they can all be easily calculated in $O(n^2)$ time.*

An overview of the reachability algorithm. We now give a first rough description of our algorithm for testing whether two given placements of L can be reached from one another by a continuous collision-free motion.

The proposed algorithm consists of two parts:

1. Building the connectivity graph (preprocessing), and
2. Searching for a path from the initial placement of L to its destination placement.

Our goal in the preprocessing part is to build a compact and space-efficient version of the connectivity graph, in the sense that (i) it (almost) does not contain nodes representing dull cells, and (ii) it contains only partial information about interesting cells.

Since our goal is to obtain an algorithm with a close-to-quadratic performance, the main difficulty we face is the handling of type II events. The problem is that in

the worst case there are $\Omega(n^2)$ criticalities of type II where each such criticality may involve up to $\Omega(n)$ faces. If we were to handle each face separately, we would end up with an algorithm having a worst-case $\Omega(n^3)$ performance. Instead, we wish to handle these $O(n)$ (and in the worst case $\Omega(n)$) faces in an implicit manner, so that each type II event can be "encoded" into our data structures in only polylogarithmic time, in a manner that still allows us to trace these changes efficiently when needed. (Note, in contrast, that type I events are "harmless"—there are $O(n^2)$ such events and each induces only $O(1)$ changes in the combinatorial structure of *FP*.)

To achieve this, we make use of the following observations, which are essential to the subsequent analysis.

Observation 1. If we *ignore the labeling* of the segments in S_θ, then a type II criticality does not change the combinatorial structure of the planar arrangement of S_θ (again we remind the reader that in this statement we ignore the effects of this criticality on the endpoints of the corresponding L_i's, which are treated separately as type I events).

Observation 2. Taking the labeling of the segments in S_θ into account, each of the $O(n)$ adjacent faces f participating in a type II critical event goes only through one of the four following combinatorial changes (where we assume that the vertical bars of two expanded obstacles L_i, L_j overlap and that L_j moves westwards relative to L_i),

(i) f changes its eastern wall from L_j to L_i (see, e.g., f_1 in Fig. 4);

(ii) f changes its western wall from L_i to L_j (f_3 in Fig. 4);

(iii) f is "squashed" between L_i and L_j, i.e., shrinks to a segment and then disappears (f_2 in Fig. 4); or

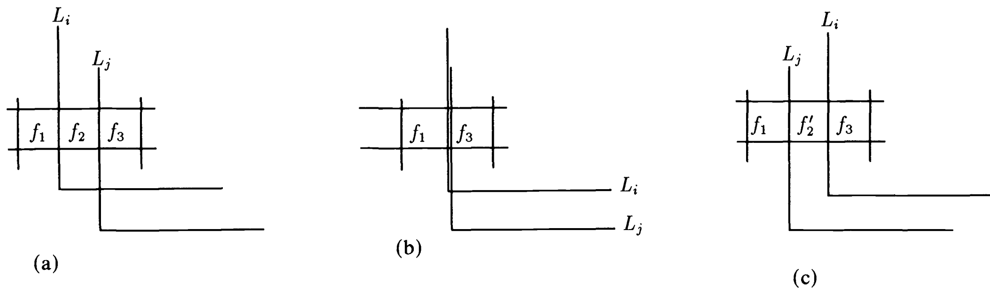


FIG. 4. *The four kinds of faces affected by a type* II *event.*

(iv) f newly appears, initially as a segment, and then expanding into a rectangle bounded between L_i and L_j (and two other horizontal bars) (f_2' in Fig. 4).

Similar changes occur when two horizontal bars of some L_i, L_j overlap.

Our idea is to take advantage of these facts for obtaining an economical representation of *FP*. Informally, we decompose *FP* into connected *subcells*. Each subcell c_f corresponds to a face f in the *unlabeled* arrangement S_θ, and consists of the union of all the slices of *FP* that correspond to f, obtained as we vary θ in a maximal interval τ in which no type I criticality affects the structure of f and no type II criticality causes f to be squashed or disappear (as in (iii) above, or its θ-symmetric counterpart (iv)). That is, $c_f = \{(X, \theta)\colon \theta \in \tau, X \in f_\theta\}$ where τ is as above and f_θ denotes the portion of

FP_θ that corresponds to f. (Note that in this context f is represented in a purely combinatorial manner, by its "location" in the unlabeled arrangement S_θ.)

Some of the subcells of FP are stored as the nodes of a *connectivity graph CG*, with edges connecting pairs of adjacent subcells, in the sense that they admit direct crossing between them either at a fixed θ through some imaginary wall, or by crossing a critical orientation (of type I) delimiting both subcells.

The main ideas of our algorithm are (i) to avoid representing dull cells as much as possible, and (ii) to store each subcell of an interesting cell as a single entity as long as it is not affected by any type I change. Thus, even though the walls of the cross-section of this subcell may change repeatedly due to type II events, we do not record these changes in the subcell. Instead, these changes are stored in a more global manner in certain auxiliary data structures and their "overall effect" is retrieved upon demand in an efficient manner.

Our algorithm proceeds roughly as follows. It starts with computing and sorting all the critical orientations. Then the FP cross-section FP_{θ_0}, for some initial noncritical orientation θ_0, is built. All the faces of FP_{θ_0} are marked *active* and the "initial layer" of the connectivity graph CG is laid, with a node corresponding to each face of FP_{θ_0} (more precisely, to each subcell of FP whose cross-section at θ_0 is that face) and edges connecting pairs of faces (subcells) adjacent along some imaginary wall. As θ increases from θ_0 and criticalities occur, we do the following:

—When a type I criticality occurs, we update the topological structure of the current FP cross-section around the critical contact, add new nodes and edges to the connectivity graph to reflect this local change, and make all the faces that are involved in the change *active*.

—At a type II critical orientation, we mark all the faces that get squashed in this event as being *inactive*. More precisely, each such face f is replaced in the current arrangement S_θ by a similar looking face f', in which a pair of opposite walls have been swapped. Even though f and f' occupy the same place in the (unlabeled) arrangement S_θ, there is clearly no direct passage between them. It is therefore important to record the fact that the corresponding face in S_θ is no longer *active*, meaning that it is as yet not connected to any *active* subcell (where a type I event occurs).

When a type I event occurs, it generates potentially new nodes (subcells) in the connectivity graph (corresponding to the faces that participate in this change). However, such a node could designate a subcell already represented by another node of CG (which has been created by a former type I event or at the initial θ_0, and which may have undergone a sequence of type II changes to reach the current face). Since these changes are not stored directly in the subcells, a major novel component of our algorithm is to determine, for a given type I contact, which faces of the unlabeled arrangement S_θ participate in it, and whether any of these faces is a portion of an already defined node of the connectivity graph. How this is achieved is described below.

More formally, we have the following definition.

DEFINITION. A face of the current cross-section FP_θ of FP is called *active* if the subcell containing it already has a representing node in CG; otherwise, the face is called *inactive*.

All the faces of FP_{θ_0} are *active* as the first step of the algorithm assigns a node in CG for every cell which has a cross-section in θ_0. A face remains *active* as long as it is not squashed by a type II event. When a face is squashed by a type II event it becomes *inactive*. An *inactive* face becomes *active* if it participates in a type I event. Consequently, all faces which are slices of interesting cells will be *active* and all faces belonging to dull cells will be *inactive* unless they have a cross-section in θ_0.

In addition, we also update, at each type II event, auxiliary global information, such as the horizontal ordering of the vertical bars of the L_i's, the vertical ordering of the horizontal bars, etc.

Reaching $\theta = \theta_0 + 2\pi$ we perform some additional work to "wrap" the connectivity graph around.

There are two additional events during the θ-sweep. These occur at the orientations θ_s and θ_d of the initial and final placements of L, respectively. The discussion of what happens at these events is postponed to the full description of the algorithm in § 4. We merely note that the introduction of these θ's enables us to locate the two nodes v_s and v_d of CG corresponding to the subcells containing the initial and final placements of L.

Since we create new nodes and edges in CG only at type I events, and each such event involves only $O(1)$ faces of FP_θ, the size of CG will be at most $O(n^2)$.

The reachability query is then treated by searching the connectivity graph for a path between v_s and v_d. If a path is found, then the answer to the query is YES, otherwise the answer is NO. As already noted, this still falls short of producing the path when one exists. See below for a discussion of this issue.

For the convenience of the analysis we assume that no two critical orientations coincide. In particular, this requires that no three obstacle points be collinear and no two pairs of obstacle points lie on parallel lines. However, such degenerate cases can be handled by an appropriate and slight modification of the algorithm.

3. Data structures. Throughout the algorithm we use the following data structures:

- CG—the connectivity graph;
- The horizontal package—retaining all the necessary information about the current cross-section of FP from a "horizontal" point of view. The package consists of:

Q_H—a balanced binary tree storing the n vertical bars of the L_i's in a left-to-right order;

R_H—a left-to-right ordered list of $n+1$ segment trees, where each tree describes (certain horizontal bars intersecting) a vertical slab of FP_θ between two lines containing adjacent vertical bars of the L_i's;

U_H—a data structure for answering queries of the form: "how many horizontal segments of S_θ are stabbed by a query vertical segment?";

- The vertical package—storing all the necessary information about the current cross-section of FP from a "vertical" point of view; it consists of three substructures Q_V, R_V, and U_V, defined in an analogous manner.

Let us now describe the structures in detail.

3.1. The connectivity graph. To better describe the structure of the connectivity graph, let us first be more precise about the imaginary walls. Four segments extend from each L_j (Fig. 5). Each segment extends until it hits some orthogonal segment:

α_j—a westwards extension of the horizontal bar of L_j;

β_j—a westward-directed segment emanating from the upper external vertex p_j of L_j;

γ_j—a southwards extension of the vertical bar of L_j; and

δ_j—a southward-directed segment emanating from the right external vertex r_j of L_j.

However, to simplify our structures we will use only the walls α_j, β_j to decompose FP_θ into faces; γ_j and δ_j will be used only in some auxiliary data structures. With this convention, each face f of FP_θ has a unique eastern wall e_f, which is a connected interval of the vertical bar of some L_j, and f consists of all points z for which there exists a horizontal segment connecting z to a point on e_f whose relative interior does

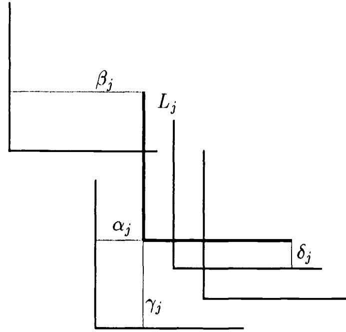
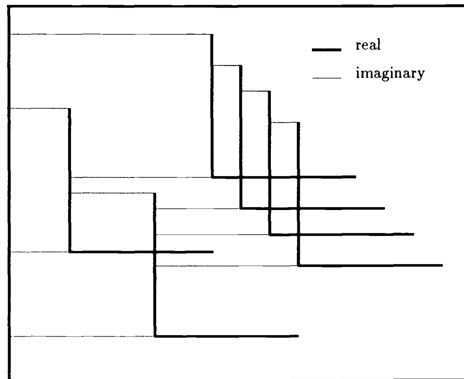
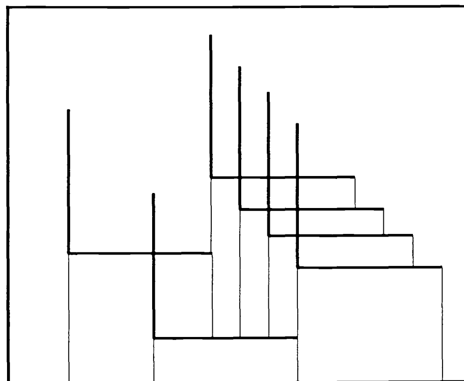


FIG. 5. *The imaginary walls.*



(a)



(b)

FIG. 6. *The "horizontal" and "vertical" planar arrangements.*

not meet any L_k (i.e., z can "see" e_f when looking directly eastwards). See Fig. 6(a) for an illustration of FP_θ.

The connectivity graph $CG=(V, E)$ has a set of nodes V and a set of edges E. Each node $v \in V$ corresponds to a subcell c of FP. New nodes are created and added to CG only when the corresponding subcells participate in a type I change. A subcell can be characterized at each orientation θ by its eastern wall (in a manner to be made more precise below). However, this information is stored in the instantaneous data structures describing FP_θ and is not encoded into the nodes of the connectivity graph. CG describes, at a certain orientation, some portion of the planar arrangement induced by $\{L_j, \alpha_j, \beta_j \mid j=1, \cdots, n\}$; we denote this extended collection of segments by S_θ^*.

The edges of CG connect adjacent subcells in FP according to the following two rules, which also give details about the changes in the corresponding faces of FP_θ and the generation of corresponding nodes in CG:

- *Neighboring θ intervals*: When a type I critical event occurs, the topological structure of FP_θ changes. For example, as a horizontal bar of some L_k hits a vertical bar l_i of another L_j while moving eastwards (Fig. 7), some face f is split into two adjacent faces f_1 and f_2. If f is *active* at the time of the split (in the particular situation depicted in Fig. 7 this will be the case, as our construction will imply), then its containing subcell c already has a representing node v in the connectivity graph. If f is not *active* (as might happen in other cases, e.g., when a vertex of an L_i first penetrates f), we add a new node to CG to represent f. Since the two subcells of FP containing f_1 and f_2 are adjacent to c and admit direct crossing between them and c, each of the two respective newly-generated nodes of CG will be connected by an edge to v (by definition, both f_1 and f_2 become *active*).

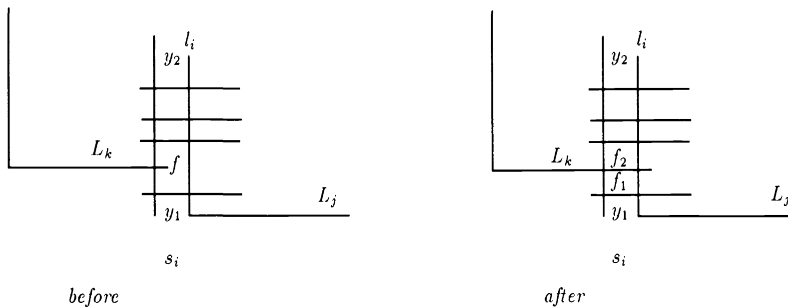


FIG. 7. *A type* I *event.*

Similarly, when the motion of L_k with respect to L_j is reversed, two faces of FP_θ are merged into a new face f (which by definition becomes *active*). Again, if any of these two faces is *active*, its representing node in CG is connected to the new node representing f in CG. (If any of the merged faces were *inactive* we do not allocate a node in CG for that *inactive* face. The connectivity of our graph is not hindered by this overpass since the subcell of FP corresponding to this *inactive* face would have been a "dead-end." If, on the other hand, either the initial placement or the final placement of L is in that subcell, its face would be activated, as we will later describe.)

Similar generation of new faces of FP_θ, corresponding nodes of CG, and connecting edges occurs when the end of the vertical bar of L_j hits the horizontal bar of another L_k, or when an endpoint of the horizontal bar of some L_j comes to lie on the horizontal bar of another L_k, or on one of the horizontal extensions α_k, β_k of another

L_k, or, similarly, for an overlapping of vertical bars. We leave the routine details of this generation to the reader.

• *Overlapping θ intervals*: The extensions α_j, β_j—the imaginary walls—do not portray real obstacles. Therefore, the two nodes of CG representing subcells that contain two faces that border on such a wall are connected in the connectivity graph by an edge. Such connections are made either in the initial layer of CG at θ_0, or at the appropriate type I event.

CG is the only structure pertaining to the three-dimensional space FP. The rest of the structures aim to represent the dynamically varying two-dimensional cross-section FP_θ of FP.

3.2. The horizontal package. The main structure in the horizontal package is R_H and it is supported by the auxiliary structures Q_H and U_H.

As mentioned before, FP_θ refers to the planar arrangement S_θ^* of the L_j's and their horizontal extensions. The horizontal package deals with this arrangement, whereas the vertical package deals with the arrangement induced by $\{L_j, \gamma_j, \delta_j \mid j = 1, \cdots, n\}$. (Figure 6 illustrates the two arrangements for the same collection of expanded obstacles. Figure 6(a) illustrates the "horizontal" arrangement and Fig. 6(b) illustrates the "vertical" arrangement.) Thus there is a slight asymmetry between these packages: while the horizontal package faithfully represents FP_θ, the vertical package assumes a more auxiliary role and represents faces of a different (though related) arrangement. The packages are designed to store the *active/inactive* status of faces. The vertical package serves a single purpose—recording "horizontal squashes." In a horizontal (or vertical) squash, a set of faces is deleted by two segments that partially overlap (and similar faces newly emerge). Since the extremal faces in the squash are treated separately, it follows that all the faces that are involved in the squash are rectangular with all walls real. Such faces have the same representation in both packages. This property will be important in the analysis to follow, as it will allow us to retrieve the *active/inactive* status of such a face by querying both packages without any ambiguity concerning the identity of the face.

There is an additional technical issue that needs to be discussed. To exploit Observations 1 and 2, made in § 2, the horizontal package represents the *unlabeled* arrangement S_θ^* of the L_j's, α_j's, and β_j's. Thus faces in this arrangement are represented only by their location in this unlabeled arrangement, with no immediate relationship to their actual location in FP_θ. The role of the auxiliary structures Q_H and U_H is to provide a mechanism for locating actual portions of FP_θ in the unlabeled arrangement S_θ^* or for identifying features of S_θ^* in the actual cross-section FP_θ. See below for more details.

3.2.1. The structure Q_H. Q_H is a balanced binary search tree which stores the left-to-right ordering of the *labeled* vertical bars of the L_j's. We update Q_H at every type II event that overlaps two vertical bars, by interchanging (in $O(\log n)$ time) the corresponding two adjacent labels.

In the following sections we describe the more elaborate structures R_H and U_H. Throughout the next section, where R_H and its usage are explained, we often refer to the structure U_H. U_H is a dynamic data structure used to report the number of segments, out of a collection of n horizontal segments, stabbed by a query vertical segment. All specific details concerning the structure U_H are postponed to § 3.2.3.

3.2.2. The structure R_H. We divide FP_θ into $n+1$ vertical slabs. The right side of a slab s_i is partially covered by the vertical bar of some L_j (or, for the rightmost slab,

by the right side of the surrounding rectangle), which we denote by l_i (the identity of L_j is not stored directly at l_i). l_i is divided into several segments, each of which is an eastern wall of some face. Recall that, by our preceding definition, a face of FP_θ is a maximal connected two-dimensional component of FP_θ whose eastern wall is a connected portion e of the eastern wall l_i of some slab s_i and contains all the points that are horizontally visible to the west from e. This connected portion e of the eastern wall of s_i is delimited on both ends either by an endpoint of the vertical bar of the corresponding L_j or by the intersection of the vertical bar of L_j with a horizontal bar of another L_k. See Fig. 8 for an illustration.

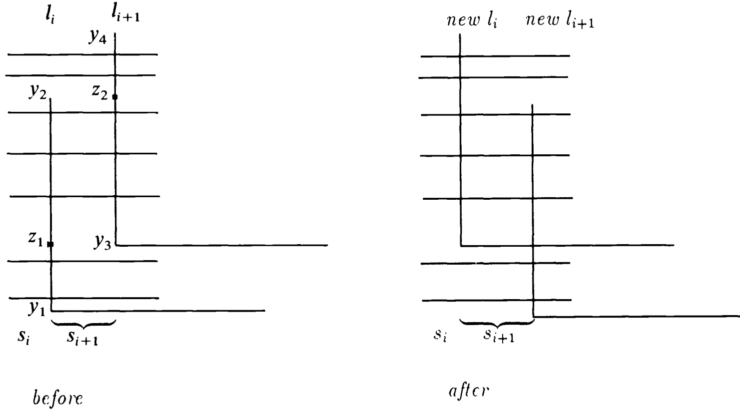


FIG. 8. *A type* II *event.*

For each slab s_i we keep in R_H information about the faces whose eastern wall is a portion of l_i. (Note that the L_j containing l_i may change during type II events, but that these changes are not stored directly in these faces.) For each such face f, we store (in part, implicitly) the following information:

(i) Is f *active* or *inactive*?

(ii) If f is *active*, then we also keep a pointer to the node in CG representing the subcell that contains f.

What happens to R_H when a type II critical event occurs? When two vertical parallel edges overlap, a set of adjacent faces belonging to some slab s_i is squashed (Fig. 8) and a new set of corresponding faces newly emerge. Suppose s_i and s_{i+1} are two adjacent slabs (we number the slabs from west to east), and let l_i and l_{i+1} denote their right delimiting vertical bars. Suppose a type II criticality (partially) overlaps l_i and l_{i+1}. At the instant of the overlap, let

$$l'_i = l_i \setminus l_{i+1}, \qquad l'_{i+1} = l_{i+1} \setminus l_i,$$

and

$$\bar{l}_i = l_i \cap l_{i+1}.$$

Then s_i and s_{i+1} should be updated by:

(i) moving the l'_i part from s_i to s_{i+1},

(ii) moving the l'_{i+1} part from s_{i+1} to s_i, and

(iii) marking the $\bar{l}_i$-faces of s_{i+1} as *inactive.*

Recall that the marginal effects of the overlap, that is, the changes in the faces neighboring the ends of $\bar{l}_i$, are treated separately as type I events. We also allow $\bar{l}_i$ to

be empty, which is the case when two vertical bars become collinear without actually overlapping. The foregoing discussion applies to this case as well.

In R_H, for each slab s_i we keep a segment tree T_i. Segment trees are a useful tool for storing and updating sets of intervals when the endpoints of the intervals are known in advance. Usually, some additional information is stored with the intervals (see [Me]). The leaves of the tree T_i correspond to atomic segments which are in our case the eastern walls of the faces of s_i, ordered from south to north. We use a segment tree, since it enables us to mark a contiguous set of elements (in our case, a contiguous set of faces) efficiently.

Now we are ready to describe how each step in handling a type II event as above is executed. We assume that each critical orientation was precomputed with some additional information about the exact location of the corresponding event. So we know the location of l_i and l_{i+1} in our (rotated) coordinate system. With this information we first search the tree Q_H to find the rank of the relevant slabs in R_H. This is easily done in time $O(\log n)$.

To compute the portions of the trees of R_H corresponding to l'_i, l'_{i+1}, and $\bar{l}_i$, we consult U_H (as explained in § 3.2.3).

To update T_i at a type II event at which the eastern sides of s_i and s_{i+1} overlap, we proceed as follows. Consider first the faces that have just become *inactive*, and let $F=\{f_j, f_{j+1}, \cdots, f_{j+m}\}$ be the contiguous sequence of these faces. We record this change in T_i so that a node w of T_i is marked as becoming *inactive* at θ if all the faces corresponding to the leaves of the subtree of w are in F and the faces corresponding to the leaves of the subtree of the father of w are not all contained in F (this corresponds to the usual way of storing a segment in a segment tree). The orientation θ at which this change occurs is also stored at w, serving as a *time stamp.* If for some $\theta_1 > \theta$ we want to mark w as again becoming *inactive*, we just update the "time" of the event, increasing it to θ_1. So each node of T_i stores only $O(1)$ information. (Note that this is a bit different from the standard usage of segment trees in which each node can store a list of segments that "cover" it. Thus our segment trees require only linear storage, in contrast to standard segment trees, which may require $O(n \log n)$ storage.) To record the inactivity of the set F in T_i, we begin by finding the ranks of f_j and f_{j+m} in T_i, using U_H. Then we search for the corresponding leaves of T_i. During the search we update the relevant nodes of T_i with the "inactivity at θ" stamp, according to the above rule. Only $O(\log n)$ nodes are updated, and the entire operation takes $O(\log n)$ time.

A complementary change occurs when some faces of S_θ^* become *active.* This happens only at events of type I (including those accompanying a type II change), and involves only a constant number of faces, which are simply marked as *active* (with the corresponding θ time stamp) in the corresponding leaves of the appropriate trees T_i. However, in this step we need to know whether any of the relevant faces is already *active*, so that we can use the same node of CG already representing that face.

To query the *active/inactive* status of a face f, we first find to what slab s_i f belongs. Since we know a point on the eastern wall of f (by the geometry of the type I event) we can obtain s_i by consulting Q_H. Next, we look for the rank of f using U_H. Then we search T_i (the tree that describes s_i) for the leaf with this rank. During the search we compute the maximum θ inactivity stamp along nodes in the search path; call this maximum $\theta_{inactive}$. When we get to the desired leaf, if it is marked *inactive*, then we conclude that the face is currently *inactive*; if the leaf is *active*, we compare the θ at which it has become *active* with $\theta_{inactive}$. The larger of these two θ values determines

whether f is *active* or *inactive*. The time needed to perform this query is $O(\log n)$. If f is *active*, we also obtain the corresponding node v of CG to which it points; we can now use v to link it to the new (*active*) subcells that appear at the current type I event. (However, some additional steps are needed in determining the *active*/*inactive* status of f—see the discussion below concerning the handling of type I changes.)

In addition to marking the *active*/*inactive* status of nodes of the trees T_i, we also have to transfer portions of one tree to an adjacent one (those corresponding to l_i', l_{i+1}' as defined above). For this purpose, instead of using balanced binary segment trees, we use 2–3 segment trees, which allow for an efficient splitting and concatenation of such trees in time $O(\log n)$ per operation. (See [KrO]; note that our case is analogous to that of "stabbing counting queries" of [KrO], since in each node we keep only $O(1)$ information, i.e., the θ-inactivity stamp.)

So far we have described the effect on the horizontal package of type II events at which two vertical bars overlap. Consider next a type II event where two horizontal bars of two expanded obstacles L_j, L_k overlap, or simply become collinear. The (unlabeled) arrangement S_θ^* undergoes several combinatorial changes, all accountable by the type I events accompanying the overlap at its endpoints, plus a "horizontal squash" of some contiguous sequence of faces (see Fig. 9). The horizontal squash is not recorded at all in R_H (but is recorded in the vertical structure R_V) and, in itself, does not effect the combinatorial structure of the unlabeled S_θ^*. However, some of the type I changes do need to be recorded in R_H. For example, in the situation depicted in Fig. 9, we need to insert a new leaf into the segment tree T_i of the slab corresponding to L_k. This leaf stands for the new face f, whose eastern wall is the lowest segment of L_k. f is marked *active*. In the symmetric situation, an extreme leaf of some T_i may have to be deleted. The effects of these type I changes on CG are described below.

How is R_H affected by a type I event? Consider, for example, a horizontal bar of some L_j hitting the vertical bar l_i of the slab s_i while moving eastwards (Fig. 7). As in type II events, using the actual geometric data accompanying this critical event, we

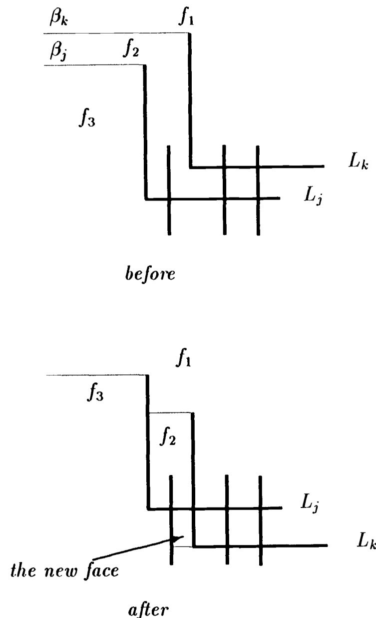


FIG. 9. *A horizontal "squash."*

can identify the corresponding slab s_i of R_H and in this slab we can find (the rank of) f_j—the face that is going to split. We then split f_j into two new subfaces, update s_i and T_i accordingly, add two new nodes to the connectivity graph to represent these two new (and *active*) faces, and point to the new nodes from the two corresponding newly-generated leaves of T_i. Since f_j was not rectangular prior to the change, it was necessarily *active*, so we connect the two new nodes of CG to the node representing f_j. By the same token, we make the new face into which the tip of L_j penetrates *active*.

Note that to determine the *active*/*inactive* status of the face f_j, it is not enough to consult R_H only, because f_j could have already become *inactive* in a horizontal squash, and such an event is marked only in R_V. A face is really *active* if and only if it is determined to be *active* in both R_H and R_V. A technical difficulty arises here, because R_V does not represent exactly the same faces as does R_H, since it stores the partition created by the L_j's and their *vertical extensions*. However, as already noted, a face that had been involved in a horizontal squash and was not reactivated later must be rectangular, with its four sides *real*, and, as such, is stored identically in both packages. We thus proceed as follows. Assume that f_j has been determined to be *active* in R_H. Find the horizontal bar passing through the top endpoint of the eastern wall of f_j (using U_H, as described in the next section). Regard that portion of that bar immediately to the left of this endpoint as the northern wall of some face f' in R_V, and query R_V to determine the *active*/*inactive* status of f'. If f' is *inactive*, then so is f_j (in which case f' and f_j represent the same region of the θ cross-section). Otherwise, f_j is *active*.

When a horizontal bar leaves a vertical bar while moving westwards (e.g., exchange *before* and *after* in Fig. 7), the steps are quite similar to those described above, and consist of checking the *active*/*inactive* status of the relevant faces in R_H and in R_V, adding nodes to the connectivity graph, adding the relevant edges to CG, etc.

Concerning the additional type I-like updating that should be done at the extreme faces of a type II overlap, consider, for example, the situation shown in Fig. 8. The face in s_i whose eastern wall contains z_1 is split into two *active* faces, with appropriate connections made in CG; similar changes apply to the face in s_{i+1} whose eastern wall contains z_2.

Finally, consider the effects of the type I changes accompanying a type II event in which the horizontal bars of L_j and L_k overlap for some j and k. We have noted above how these changes effect R_H. Their influence on CG has to reflect the possible changes in the combinatorial structure of the faces bordering the imaginary horizontal extensions $\alpha_j, \alpha_k, \beta_j, \beta_k$. For example, in the situation shown in Fig. 9, before the overlap we have a face f_2 bounded by β_j from below and by β_k from above and connected, via these extensions, to the face f_1 above it and to the face f_3 below. After the overlap, f_3 becomes directly connected to f_1. To keep track of these changes, we create new nodes in CG to represent f_1, f_2, and f_3 after the change, connect each node to the corresponding old node, and add edges connecting the new f_1 to the new f_2 and to the new f_3. Similar action is taken to handle the possible overlap between the extensions α_j, α_k. (Note, however, that these changes take place only if the extensions β_j, β_k (or α_j, α_k) actually overlap. If β_k ends on a vertical bar of another L_t that lies to the right of the vertical bar of L_j, no changes are required.)

3.2.3. The structure U_H. U_H is a structure for solving the following problem: *Let H be a set of n horizontal bars (of the L_j's), and let s be a query vertical line segment; report the number of segments in H that are intersected by s.* The structure U_H that we use is a *dynamic* version of a well-known two-level combination of a primary segment

tree and auxiliary range trees (as in [Ov], for example). More specifically, we project all the segments in H on the x-axis. We construct a segment tree P for the intervals obtained. At each internal node ν of P we store in a binary tree M_ν the segments that are assigned to ν, ordered by y-coordinate. M_ν is a one-dimensional range tree. The structure uses $O(n \log n)$ storage. Let $s = \overline{(x, y_1)(x, y_2)}$ be a query vertical segment. We search in P for x. For each node ν on the search path we count how many segments in M_ν lie between (x, y_1) and (x, y_2); summing up all the counts, we get the number of line segments stabbed by s. The query time is $O(\log^2 n)$, because we perform $O(\log n)$ searches in the auxiliary trees. The structure is dynamic and can be updated in $O(\log^2 n)$ time [Ov].

To compute the portions l'_i, l'_{i+1}, and $\bar{l}_i$ mentioned earlier, we consult U_H. For example, let us see how the portion l'_i of Fig. 8 is computed. It is delimited by the lower endpoints of the vertical bars of the two participating L_j's. Using U_H we can easily determine the number of horizontal segments stabbed by the segment l'_i. This number, increased by 1, is also the rank of the face of s_i at which we should split the segment tree T_i into portions corresponding to l'_i and $\bar{l}_i$. In a similar way, we compute the portion l'_{i+1}, and as a result of these computations we also have available the $\bar{l}_i$-parts of s_i and s_{i+1}. This takes $O(\log^2 n)$ time.

The structure should be updated at each type II event that overlaps two vertical bars and at each type I event in which an endpoint of one horizontal bar meets another vertical bar, because then the horizontal order of the endpoints of the horizontal bars in H changes. Each such update costs $O(\log^2 n)$; for details, see [Ov]. A type II event that overlaps two horizontal bars changes the vertical ordering of the horizontal bars; this does not effect the counting, but since we want to take some additional advantage of U_H (to be described below), we update U_H upon such events as well.

Most of the time we employ the *unlabeled* arrangement. There are, however, occasions when we have to resolve the geometric anonymity of the segments in that arrangement. In some of these cases it is sufficient to consult the binary search trees Q_H, Q_V; in other cases (see the previous subsection) we have to allow for queries of the following kind: "Given a point z in the current cross-section of *FP*, which is the nearest segment (corresponding to some horizontal bar) above z that is vertically visible from z?" Such queries can be answered using U_H. Let $z = (x_1, y_1)$. We search in P for x_1. In each node ν along the search path, we search in the range tree M_ν for the lowest segment that is higher than y_1, and the lowest of these $O(\log n)$ candidates is the desired segment. This procedure takes $O(\log^2 n)$ time. The orthogonal type of such a "ray shooting" query, i.e., finding the nearest eastern bar horizontally visible from a point, can be answered in a completely symmetric manner by consulting U_V.

3.3. The vertical package. We mentioned before that the horizontal package and the vertical package each describes a different planar arrangement. They coincide, though, in the description of (the rectangular) faces which become *inactive*, are *inactive*, or turn from *inactive* to *active*, as follows from the discussion in the previous subsection. The vertical package is handled in a manner completely symmetric to that of the horizontal package with regard to the respective unlabeled planar arrangement (namely, the L_i's and their vertical extensions); we thus omit the details of the manipulation of this package.

3.4. Summary of operations at events of each type. We conclude this section with a summary of operations on the data structures taken at each type of events. For events of type I we describe the operations as they are performed at the event depicted in Fig. 7 (other kinds of type I events are handled symmetrically):

• When the right endpoint of L_k penetrates f, we identify f in the corresponding tree T_i of R_H by first querying Q_H with the information accompanying the event.
• We then find f in T_i using U_H.
• To decide whether f is *active* or *inactive*, we query both R_H and R_V, finding f in R_V requires "ray shooting" using U_H.
• If f is *inactive*, we make it *active* and add a new node to the connectivity graph pointed to from the leaf corresponding to f in T_i.
• When the right endpoint of L_k reaches the western wall of f we replace f in T_i by two faces f_1 and f_2 (inserting a new *active* "key" to T_i).
• Finally, we update CG by adding two new nodes corresponding to the subcells containing f_1 and f_2 and connecting them with edges to the node representing f.

The operations at a type II event are described for the event depicted in Fig. 8:
• We find the rank of the slabs s_i and s_{i+1} and the corresponding trees T_i and T_{i+1} in R_H using Q_H.
• Using U_H we compute the portions of T_i and T_{i+1} corresponding to l'_i, l'_{i+1}, and $\bar{l}_i$.
• We transfer the l'_i portion from T_i to T_{i+1} and the l'_{i+1} portion from T_{i+1} to T_i.
• Finally, we mark the $\bar{l}_i$ portion of T_{i+1} as *inactive*.

4. Algorithmic details and complexity analysis. In this section we complete the details of the algorithm, prove its correctness, and analyze its time and space requirements.

4.1. The algorithm.

Construction of FP_{θ_0}. First, the critical orientations are computed and sorted. Then a noncritical θ_0 is chosen and the Minkowski differences L_i for θ_0 are computed. We sort the vertical bars of the L_i's according to their x-coordinate and store this ordering in Q_H. Now we start sweeping a vertical line across the plane from left to right while maintaining a sorted list F of all the horizontal line segments intersecting the line being swept. Each time we sweep across an internal vertex q_j of some L_j, we construct a new segment tree T_i that describes the slab s_i. To build T_i we first locate the position of p_j (the upper external vertex of L_j) and of q_j in F and then allocate a segment tree for the intervals in F from p_j to q_j. We mark all the faces *active*. Each time we sweep across a right external vertex r_j of some L_j, we remove the segment $\overline{q_j r_j}$ from F. The last stop of the sweep is the eastern wall of the surrounding rectangle, where we have exactly one face which we will keep in T_{n+1}. The set of all the segment trees $\{T_i \mid i = 1, 2, \cdots, n+1\}$ constitutes R_H. Similarly, we construct Q_V and R_V.

During the line sweep we also lay an initial layer of CG. Every time we create a segment tree T_i, we add a node to CG for every leaf of T_i. These nodes represent all the faces whose eastern wall is in l_i (l_i, the vertical bar on the right side of s_i). Each pair of nodes whose corresponding faces share an imaginary horizontal wall is connected by an edge. Note that at least one node of such a pair is an extreme face of some T_i. To obtain the desired connections, we maintain a list W of all corners of L_j's that have already been swept through and that are still "visible" from the sweep line. Whenever we sweep through a new vertical bar l_i, we remove from W all endpoints horizontally visible from l_i, and connect the extreme nodes of the corresponding segment trees to the appropriate new faces of s_i. The endpoints of l_i are then added to W.

Overall, this initial phase of the algorithm requires $O(n^2)$ time, since the sweep itself is easily seen to require $O(n \log n)$ time, and for each slab we build a segment

tree when the endpoints of the segments are already sorted by the sweep structure.

The second phase of the algorithm, updating the structures as the orientation changes, is discussed in detail in § 3. A missing link there, though, is the wrap-around of *CG*. Reaching $\theta = \theta_0 + 2\pi$, our task is to identify the nodes of the *active* faces with their matching peers in the first layer of *CG*. But at $\theta_0 + 2\pi$ there is no longer a way to know to which face in $FP_{\theta_0+2\pi}$ a node of the first layer of *CG* belongs. A simple solution to the problem is to keep a duplicate copy of the horizontal package at θ_0. Getting to $\theta_0 + 2\pi$ we scan the current updated R_H and for every *active* face of $FP_{\theta_0+2\pi}$, we search for its matching (identical) face in the original version of R_H and identify the two corresponding nodes in *CG*.

Some final details of the processing involve the reachability query itself. Let $Z_s = (X_s, \theta_s)$ be the initial placement of *L*, and $Z_d = (X_d, \theta_d)$ be the final placement. Let c_s and c_d be the subcells of *FP* containing Z_s and Z_d, respectively. To ensure that these subcells will be represented in *CG*, we add two artificial critical events, θ_s and θ_d, during the θ-sweep. The purpose of these events is to identify or otherwise introduce the nodes v_s, v_d of *CG* corresponding to c_s, c_d, respectively. When we reach θ_s we look for the face f_s containing X_s, using first U_V to identify the nearest vertical bar to the east of X_s that is horizontally visible from X_s, and then U_H to find (the rank of) f_s in the corresponding slab. If f_s is *active* we obtain from it a pointer to v_s; otherwise, we make it *active*, update R_H, R_V accordingly, create a new node v_s in *CG* to represent f_s, and keep a pointer to this newly-generated node. Similar steps are taken when we reach θ_d. After the completion of the θ-sweep, we search for a path in *CG* between v_s and v_d. If a path is found, the algorithm outputs YES; otherwise it outputs NO.

The following proposition justifies the reduction of our motion-planning problem to the purely combinatorial path searching through *CG*.

PROPOSITION 4.1. *If both Z_s and Z_d are free positions of L, then there is an obstacle-avoiding motion between Z_s and Z_d if and only if v_s and v_d belong to the same connected component of CG.*

Proof. For the "if" part, let ϕ be a path between v_s and v_d in *CG*. First, it is easily verified that any single node v of *CG* represents a connected portion c of *FP*, that is, any two placements of L within c can be reached from one another along a collision-free path that remains in c (see, also, Remark 4.1 below). Next, each edge e along ϕ represents one of the following "crossings":

(i) e connects two nodes representing subcells whose cross-sections at some θ are adjacent along some imaginary wall; in this case there is a direct translational crossing of L at this θ between the subcells (although this is not required in this part of the proof, we note that our construction ensures that if this crossing is possible at one θ, it is possible at all θ's at which both subcells exist); or

(ii) e connects nodes representing subcells that were both influenced by the same type I event; in this case it is easily verified that there is some rotational crossing between the two relevant subcells. (There is one exception: If these subcells represent two faces that were split from one *inactive* subcell, our procedure has created a shortcut connection between them, which does not correspond to direct crossing between the subcells. Nevertheless, it is still possible to cross from one of them to the other by passing through the *inactive* subcell adjacent to both.)

These observations clearly imply that the given path in *CG* can be transformed into a collision-free path within *FP*.

As for the "only if" part, define a retraction-like mapping $\Psi_H : FP \to BFP$ as follows. For each $Z = (X, \theta) \in FP$ move the object L by translating it in the direction of its "horizontal" bar $\overrightarrow{rq}$ (so that r moves "towards" q) until the vertical bar hits an

obstacle (or L reaches the enclosing rectangle). The resulting position is $\Psi_H(Z)$. (Even if the horizontal bar touches an obstacle at Z, we still allow this sliding motion to be performed.) It is easily checked that Ψ_H is continuous in the interior of FP except at points that lie on the imaginary extensions α_j, β_j for some L_j.

Now suppose there is a collision-free path $F:[0,1]\to FP$ connecting $Z_s=F(0)$ to $Z_d=F(1)$. The composition $G=\Psi_H\circ F$ is a piecewise-continuous path which, using standard topological arguments akin to those used in [SS], we can assume to consist of only a finite number of connected pieces. Note that the endpoints of each piece of G lie in *active* faces of the corresponding cross-sections of FP, because each endpoint is either Z_s or Z_d, or lies on an imaginary horizontal extension, which, by construction, always bounds two *active* faces. Moreover, by construction of CG, for any two consecutive portions of G, the nodes of CG in which the first subpath ends and in which the second subpath begins are connected by an edge. It therefore suffices to prove that each subpath G' of G induces a path in CG connecting the two nodes that contain the endpoints of G'.

Note that, in the cross-sectional representation of FP that we use, each path G' is represented by a point w varying continuously along an eastern wall of some face(s) of the arrangement S^*_θ (that also varies with θ). Since w begins its motion in an *active* face, it suffices to verify that it always remains in an *active* face, and that whenever this face changes, a corresponding type I event which involves this change occurs (and induces a connecting edge between the corresponding nodes of CG). To show this, we first break G' into a number of pieces, such that on each of them θ varies monotonically and does not cross θ_0; without loss of generality we can assume that there are only a finite number of such pieces. If such a piece starts at an *active* face and proceeds in the direction of increasing θ (including the case of crossing $\theta_0+2\pi$ back to θ_0), then our construction is easily seen to imply the property asserted above. (A point to note here is that if w crosses between faces through an imaginary extension, then this crossing must have been possible at a type I event that involved both faces, and therefore created the corresponding edge in CG.) If θ decreases along such a subpath, we have to be more careful, as our construction does not guarantee that the *active* status is propagated backwards in θ. However, a close inspection of our construction shows that if we move backwards in θ from an *active* face f to an *inactive* face f', then f' must be a "dead-end" face that will eventually (i.e., if we continue to decrease θ) be squashed at some type II event. Since G' ends in an *active* face, it cannot stay in f' and must exit by reversing its θ direction and cross back from f' to f, or perhaps from f' to another face f'' separated from f by an imaginary horizontal extension. In the first case, we simply ignore the excursion of G' into f'; in the second case, our construction induces a shortcut connection between the nodes of CG containing f and f'', respectively. Finally, if a subpath of G' starts at θ_0 (it does so in an *active* face by construction), and moves backwards from θ_0 (θ decreasing), then it might enter an *inactive* face in $FP_{\theta_0+2\pi}$, because at this cross-section not all the faces are necessarily *active*. But the above arguments that the path will then have to move back into θ_0 apply to this case as well. This completes the proof of the "only if" part of the proposition. □

Remark 4.1. It is instructive to describe a canonical path in FP that corresponds to a given path ϕ in CG. The desired path Π is a concatenation of subpaths, each describing a simple motion of L, as follows. Suppose ϕ has reached a node v of CG and let v' be the next node along ϕ. Let c, c' be the corresponding subcells of FP. Inductively, suppose Π has already reached some placement $Z\in c$. To continue Π to reach a placement $Z'\in c'$, we proceed as follows. By construction, c is a subcell in

which θ varies between two type I critical orientations $\theta_1 < \theta_2$ and for each $\theta_1 < \theta < \theta_2$, $c \cap FP_\theta$ corresponds to the same face f in the unlabeled arrangement S^*_θ. If f contains (on its boundary) a corner w of some L_j, we move (translationally) to w, and stay at w as θ varies. Otherwise, f must be rectangular, with all four walls real. In this case we move to, say, the northeast corner of f and stay there as θ varies. Note that in the latter case the combinatorial complexity of the motion within c can be $\Omega(n)$, because the northeast corner of f can change whenever its eastern wall or northern wall changes due to a type II event.

In the physical space, the first type of motion becomes a rotation of L, while one of its three corners touches an obstacle. The second type of motion is a "gliding" motion of L, in which it rotates while each of its bars touches an obstacle, such that whenever any of the bars encounters a new obstacle o_i, the gliding continues with that bar touching o_i. Note that the middle corner q of L traverses a circular arc during each portion of the gliding.

Finally, the crossing from c to c' is easy to accomplish. If this crossing is through an imaginary wall at some θ, we move within c, as specified above, to θ (as noted above, any θ at which both c and c' exist will do) and then translate to c' through the wall. If the crossing is through a type I change, we reach the corresponding extreme critical orientation θ as above, then cross locally according to the nature of the type I change (rotating further into the new cell, translating across an imaginary wall, rotating and translating back and forth to realize an indirect connection, etc.).

4.2. Complexity analysis.

4.2.1. Analysis of the two-dimensional data structures. In this section we summarize the computational cost of maintaining and manipulating the two-dimensional data structures. We analyze below the cost of the horizontal package but the analysis of the vertical package is essentially identical.

Q_H, the structure retaining the horizontal ordering of the vertical bars of the L_i's, is a balanced binary tree with n elements. Its initial construction takes $O(n \log n)$ time. Upon each type II criticality that overlaps vertical bars, we interchange these two adjacent elements in $O(\log n)$ time. Upon each type I criticality bringing an endpoint of a horizontal bar to cross a vertical bar, we query Q_H in $O(\log n)$ time. Therefore the usage of Q_H costs $O(n^2 \log n)$ time. Being a balanced binary tree with n elements, it demands $O(n)$ space. To summarize, we have the following lemma.

LEMMA 4.1. *Q_H requires $O(n^2 \log n)$ time and $O(n)$ space, and so does Q_V.*

As to R_H, initially we build n 2-3 segment trees, as part of the initial sweep; this construction requires $O(n^2)$ time (as noted before). Afterwards, each operation on any of these trees—delete, insert, concatenate, split, or query—requires $O(\log n)$ time. The number of operations on R_H that are required at each criticality is bounded by a constant. Thus, the usage of R_H costs $O(n^2 \log n)$ time. See [KrO] for more details. Each tree requires $O(n)$ storage, summing up to $O(n^2)$ storage for R_H. Thus, we have Lemma 4.2.

LEMMA 4.2. *R_H and R_V each requires $O(n^2 \log n)$ time and $O(n^2)$ space.*

U_H is a two-level combination of a primary segment tree and auxiliary range trees. It is built in $O(n \log n)$ time. Each update requires $O(\log^2 n)$ time and a query takes $O(\log^2 n)$ time. There is a constant number of updates and queries per criticality, so the operations on U_H take $O(n^2 \log^2 n)$ time in total. It uses $O(n \log n)$ space. Lemma 4.3 follows.

LEMMA 4.3. *U_H requires $O(n^2 \log^2 n)$ time and $O(n \log n)$ space, as does U_V.*

For details on the dynamic segment-tree–range-trees combination, see [Ov].

4.2.2. Analysis of the overall complexity. The initial phase of the algorithm (at θ_0) includes a vertical sweep and a horizontal sweep, both fairly standard and taking only $O(n \log n)$ time, as is easily seen.

How complex is CG?

LEMMA 4.4. *CG has $O(n^2)$ nodes and edges.*

Proof. At the initial phase, FP_{θ_0} has at most $O(n^2)$ *active* faces. So the first layer of CG has $O(n^2)$ nodes. During the θ-sweep we have $O(n^2)$ stops. At each stop we add a constant number of nodes to CG and with each new node we add at most a constant number of edges to connect it to some previously existing, or newly-generated, nodes. □

After the completion of the θ-sweep, we identify the nodes of CG corresponding to the *active* faces of $FP_{\theta_0+2\pi}$ with the matching nodes of faces of FP_{θ_0}.

LEMMA 4.5. *The wrap-around of CG requires $O(n^2)$ time.*

Proof. To identify nodes of the two layers, we follow the R_H structure at $\theta_0+2\pi$ and the reserved duplicate of the R_H structure at θ_0. For each *active* face of the newer R_H, we identify its node in CG with that of the first layer. Since we pass sequentially (e.g., in inorder on every tree) through the $O(n^2)$ faces, the traversal requires $O(n^2)$ time. □

Finally, to find a path from v_s to v_d (if one exists), we search through CG.

LEMMA 4.6. *The search through CG for a path from v_s to v_d takes $O(n^2)$ time.*

Proof. The search can be carried out using breadth-first-search, which is linear in the number of edges in the graph. Since CG has $O(n^2)$ edges, the bound follows. □

We are now ready for the main theorem.

THEOREM 4.1. *The algorithm answers the reachability query correctly, using $O(n^2 \log^2 n)$ time and $O(n^2)$ space.*

Proof. The correctness of the algorithm follows from the analysis in the proof of Proposition 4.1.

As for the time required by the algorithm, we start by computing and sorting the critical orientations. This can easily be done in $O(n^2 \log n)$ time. The vertical and horizontal sweeps take $O(n \log n)$ time. The building and usage of the two-dimensional data structures take $O(n^2 \log^2 n)$ time (this follows from Lemmas 4.1, 4.2, and 4.3). The wrap-around of CG requires $O(n^2)$ time (Lemma 4.5) and the search through CG requires $O(n^2)$ time (Lemma 4.6). We see, then, that the usage of the two-dimensional data structures (specifically, the usage of U_H and U_V) dominates the time complexity of the algorithm, which is $O(n^2 \log^2 n)$.

As for the storage requirements, CG consists of $O(n^2)$ elements (Lemma 4.4), and no other structure or procedure requires more space. □

5. Conclusion. We have presented here an $O(n^2 \log^2 n)$-time algorithm for the solution of the reachability problem for an L-shaped object moving amidst n point obstacles in the plane. In this section we summarize the new ideas of our approach, assess the algorithm efficiency, and point out possible extensions of this work.

The main innovation of our approach is the condensation of a potentially $\Omega(n^3)$-size configuration space FP into quadratic-storage structures using near-quadratic time. This is achieved by building a skeletal connectivity graph which, in contrast with previously suggested connectivity graphs, (almost) does not contain dull cells and suppresses the explicit representation of most of the changes that occur in interesting cells. The compaction of the connectivity graph is enabled by some auxiliary evanescent data structures which, at every instant of the θ-sweep, store necessary information about the momentary FP cross-section in an implicit compact manner.

As already mentioned, the complexity of the entire FP in our case can be $\Theta(n^3)$ in the worst case. However, in most cases one does not need to calculate the entire FP, but only its connected component C containing the initial given placement Z_s of L. Indeed, as long as L moves in a collision-free manner from Z_s, it will have to remain in C. We would like to precalculate only the component C rather than the entire FP, and to show that this will always achieve better performance than the naive cubic worst-case bound for the entire FP. This goal was achieved, to a satisfactory extent, for the general motion-planning problem with two degrees of freedom [GSS]. Unfortunately, in problems with three degrees of freedom (such as ours) this goal appears to be much harder. In the special case in which the surface patches bounding FP are n triangles in three-space, it was shown in [AS] (see also [AA]) that the complexity of a single cell in the complement of their union is at most roughly $O(n^{7/3})$. However, in the case of the L-shaped body, the surface patches bounding FP are more complicated. Nevertheless, the three-dimensional arrangement induced by these patches are analyzable with tools similar to those used in [AS], and in a companion paper [HS] we obtain a bound $O(n^{5/2})$ on the complexity of all the three-dimensional cells that contain a portion of the one-dimensional boundary of any surface in their closure, which is also an upper bound on the complexity of any single cell in these arrangements.

The reachability problem only decides whether there exists a continuous motion between source and destination. Naturally, we would like to produce such a motion if it exists, and do so in subcubic time. In [HS] we also elaborate the reachability algorithm devised in this paper into a find-path algorithm using the combinatorial result mentioned above. The find-path algorithm is based on extending the connectivity graph by recording wall changes in interesting cells while continuing to ignore most of the dull cells.

Remark 4.1 shows that the portions of the required path that lie in faces that contain corners of some L_j's are easy to produce—we simply "take a ride" on the corresponding corner. The difficulty lies in producing those portions of the path that traverse "tube-like" subcells having a real rectangular cross-section (which, however, can undergo $\Omega(n)$ changes due to type II criticalities). It is interesting to note that from a pragmatic point of view this issue may not be problematic. Assuming our object L to be equipped with tactile sensors all around it, traversing a tube-like subcell is easy to accomplish by executing the corresponding gliding motion (as in Remark 4.1) and using tactile feedback to tell when one of the two obstacles touched by L during the gliding has to be replaced by another. Thus, in this pragmatic setting, our reachability algorithm can easily be adapted to produce the desired path, at no additional overhead.

There is a close connection between the problem of the nonconvex body moving among polygonal obstacles and the problem of moving certain kinds of planar robot arms with three degrees of freedom (such as a standard three-link anchored planar arm or the "telescopic" arm studied in [AO]) in the same setting. A first step in exploiting this observation is taken in [HS].

A possible direction for further research is to check whether our algorithm is extensible to polygonal obstacles (not only point obstacles) and to an arbitrary rectilinear nonconvex moving object, without severely increasing the time and space complexity. We are presently investigating these extensions.

Finally, there is the issue of improving the performance of our reachability algorithm. Two problems suggest themselves. First, can we improve the time required by the algorithm to $O(n^2 \log n)$? As noted, the only step which requires $O(n^2 \log^2 n)$ time is the handling of the structures U_H, U_V. Second, can one show a lower bound $\Omega(n^2)$ on the number of nodes of CG that a path must traverse between some pair of

placements? For our reachability problem such a bound would not necessarily preclude the possibility of a faster decision procedure, but it would be a strong indication that quadratic complexity is probably a correct worst-case bound.

REFERENCES

[AA] P. K. AGARWAL AND B. ARONOV, *Counting facets and incidences*, manuscript, 1990.

[AO] B. ARONOV AND C. Ó'DÚNLAING, *Analysis of the motion-planning problem for a simple two-link planar arm*, Tech. Report 314, Courant Institute of Mathematical Sciences, New York University, New York, 1987.

[AS] B. ARONOV AND M. SHARIR, *Triangles in space, or building (and analyzing) castles in the air*, Combinatorica, 10 (1990), pp. 137–173.

[ABF] F. AVNAIM, J. D. BOISSONNAT, AND B. FAVERJON, *A practical exact motion planning algorithm for polygonal objects amidst polygonal obstacles*, in Proc. IEEE Internat. Conference on Robotics and Automation, 1988, pp. 1656–1661.

[Ca] J. CANNY, *The complexity of robot motion planning*, Ph.D. thesis, Computer Science Department, Massachusetts Institute of Technology, Cambridge, MA, May 1987.

[CK] L. P. CHEW AND K. KEDEM, *High-clearance motion planning for a convex polygon among polygonal obstacles*, Tech. Report 184/90, The Eskenasy Institute of Computer Science, Tel-Aviv University, Tel-Aviv, Israel, 1990.

[GSS] L. GUIBAS, M. SHARIR, AND S. SIFRONY, *On the general motion planning problem with two degrees of freedom*, Discrete Comput. Geom., 4 (1989), pp. 491–521.

[HS] D. HALPERIN AND M. SHARIR, *Improved combinatorial bounds and efficient techniques for certain motion planning problems with three degrees of freedom*, in Proc. Second Canadian Conference on Computational Geometry, 1990, pp. 98–101.

[KeO] Y. KE AND J. O'ROURKE, *Lower bounds on moving a ladder in two and three dimensions*, Discrete Comput. Geom., 3 (1987), pp. 197–218.

[KLPS] K. KEDEM, R. LIVNE, J. PACH, AND M. SHARIR, *On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles*, Discrete Comput. Geom., 1 (1986), pp. 59–71.

[KS1] K. KEDEM AND M. SHARIR, *An efficient algorithm for planning collision-free translational motion of a convex polygonal object in two-dimensional space amidst polygonal obstacles*, in Proc. First ACM Symposium on Computational Geometry, 1985, pp. 75–80.

[KS2] K. KEDEM AND M. SHARIR, *An efficient motion planning algorithm for a convex polygonal object in two-dimensional polygonal space*, Discrete Comput. Geom., 5 (1990), pp. 43–75.

[KrO] M. VAN KREVELD AND M. H. OVERMARS, *Concatenable segment trees*, in Proc. Symposium on Theoretical Aspects of Computer Science, 1989, Lecture Notes in Computer Science 349, Springer-Verlag, Berlin, New York, 1989, pp. 493–504.

[LS] D. LEVEN AND M. SHARIR, *An efficient and simple motion planning algorithm for a ladder moving in two-dimensional space amidst polygonal barriers*, J. Algorithms, 8 (1987), pp. 192–215.

[LW] T. LOZANO-PEREZ AND M. WESLEY, *An algorithm for planning collision-free paths among polyhedral obstacles*, Comm. ACM, 22 (1979), pp. 560–570.

[Me] K. MEHLHORN, *Data Structures and Algorithms* 3: *Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[Ov] M. H. OVERMARS, *Geometric data structures for computer graphics*: *An overview*, in Theoretical Foundations of Computer Graphics and CAD, R. A. Earnshaw, ed., Springer-Verlag, Berlin, Heidelberg, 1988, pp. 167–184.

[SS] J. T. SCHWARTZ AND M. SHARIR, *On the "Piano Movers" Problem* I. *The case of a two-dimensional rigid polygonal body moving amidst polygonal boundaries*, Comm. Pure Appl. Math., 36 (1983), pp. 345–398.

[SiS] S. SIFRONY AND M. SHARIR, *An efficient motion planning algorithm for a rod moving in two-dimensional polygonal space*, Algorithmica, 2 (1987), pp. 367–402.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 24-32, February 1992
© 1992 Society for Industrial and Applied Mathematics
002

ON THE AVERAGE SIZE OF THE INTERSECTION OF BINARY TREES*

R. BAEZA-YATES†, R. CASAS‡, J. DÍAZ‡, AND C. MARTÍNEZ‡

Abstract. The average-case analysis of algorithms for binary search trees yields very different results from those obtained under the uniform distribution. The analysis itself is more complex and replaces algebraic equations by integral equations. In this work this analysis is carried out for the computation of the average size of the intersection of two binary trees. The development of this analysis involves Bessel functions that appear in the solutions of partial differential equations, and the result has an average size of $O(n^{2\sqrt{2}-2}/\sqrt{\log n})$, contrasting with the size $O(1)$ obtained when considering a uniform distribution.

Key words. average complexity, binary search trees, generating functions

AMS(MOS) subject classification. 68Q25

1. Introduction. This work constitutes a contribution to the study of the formal properties of the probability model associated to binary search trees, from now on denoted bst-model. Our first motivation has been to investigate the kind of analysis underlying the obtention of statistics under the bst-model. This is the reason why we selected a problem that yields a very elementary development and solution, when considering the uniform probability model. In contrast, the use of the bst-model introduces a partial differential equation which, using Riemann's method, has a solution in terms of Bessel functions. These results reflect the structure of the model dealing with pairs of trees, and can thus be generalized to similar problems.

A great amount of work has been done on statistics for binary search trees. Most of this work relates to the average-case analysis of algorithms associated with the manipulation of this particular data structure [Knu73]. Some other works use the bst-model for computing characteristics of binary trees. Devroye proved that the average height of binary trees under the bst-model is asymptotically $O(\log n)$ [Dev86]. This result differs from the average height of binary trees under the uniform model, which is $O(\sqrt{n})$ [FO82]. This difference lies on the fact that the bst-model tends to assign higher probability to the more balanced binary trees, and relatively lower probability to the narrower trees of the same size.

We have chosen to study the average size of the intersection of two binary trees because of its simplicity. Nevertheless, this computation appears in a natural way in the analysis of a number of algorithms, for example, in processes involving tree matching [SF83] or unification [CDS89]. For instance, the intersection of binary trees is exactly the kernel of the Tree Shuffle algorithm described in [CKS89]. The time complexity of Shuffle for any pair of binary trees is twice the size of the intersection of the two trees plus one. Under the uniform model, the average size of the intersection of two trees tends to the constant 1.5, when the total size of the two trees tends to infinity. Under the bst-model this average size turns out to be $O(n^{2\sqrt{2}-2}/\sqrt{\log n})$.

The rest of this paper is structured as follows. In § 2 we introduce the basic definitions and notation. In particular we give a recursive formulation of probabilities in the bst-model, which makes the computation easier. In § 3 we obtain and solve a

---

* Received by the editors February 7, 1990; accepted for publication (in revised form) February 20, 1991. This research was supported by the ESPRIT BRA program of the EC under contract 3075, project ALCOM.

† Departamento de Ciencias de la Computación, Universidad de Chile, Casilla 2777, Santiago de Chile.

‡ Departament de Llenguatges i Sistemes, Universitat Politècnica Catalunya, Pau Gargallo 5, 08028-Barcelona, Spain.

partial differential equation which defines the probabilistic generating function associated with the size of the intersection of binary trees. In § 4 we derive exact expressions for the nth coefficient of the function obtained in the previous section, while in § 5 we deduce the main result of the paper, which is the asymptotic behaviour of this coefficient.

2. A recursive definition of the bst-model. Recall that a *binary search tree* is a data structure which consists in a binary tree whose nodes are labeled in increasing order from left to right. Binary search trees have the following recursive definition [VF90]: Given a sequence on n keys $S=(k_1, k_2, \cdots, k_n)$, where the keys belong to a totally ordered set, we define recursively the binary search tree of S:

$$BST(S)=\begin{cases}\langle BST(S_i), k_1, BST(S_s)\rangle & \text{if } |S|\geqq 1,\\ \langle\square\rangle & \text{otherwise,}\end{cases}$$

where $|S|$ denotes the number of keys in S; S_i and S_s denote the subsequences of S formed, respectively, by the elements of S which are less than k_1 and greater than k_1; and $\square$ denotes the empty binary tree.

In the model of probability associated with binary search trees, each sequence S is obtained by consecutively sampling at random n elements from a real interval, or equivalently, as far as relative ordering is concerned, the elements form a random permutation of size n. In any case, all the sequences with the same size n have the same probability $1/n!$

Let $N(T)$ denote the number of sequences S of size n that generate the same binary search tree $T=BST(S)$. Given a binary tree T, we shall denote by $|T|$ the number of internal nodes of T, and if $|T|>0$, let T^r and T^l be, respectively, the right and left subtrees of the root of T. It is shown in [Knu73, § 6.2.2, exercise 5] that we can compute $N(T)$ from the following recursive equation:

$$N(T)=\begin{cases}1 & \text{if } T=\square,\\ N(T^l)\cdot N(T^r)\cdot\dfrac{(|T|-1)!}{|T^l|!\cdot|T^r|!} & \text{otherwise.}\end{cases}$$

If we denote by $p(T)$ the probability of tree T, we have that

$$p(T)=\frac{N(T)}{|T|!},$$

therefore,

$$p(T)=\begin{cases}1 & \text{if } T=\square,\\ \dfrac{p(T^l)\cdot p(T^r)}{1+|T^l|+|T^r|} & \text{otherwise.}\end{cases}$$

The recursive manner in which we express this probability distribution is very handy to simplify some proofs about average behaviour of binary search. It allows us to split the generating functions defined from this probability.

The problem we are going to present involves pairs of binary search trees, so we must extend the probability model to pairs of trees. We are confronted with two situations:

If each of the trees is drawn *independently* of the other, the probability of the pair is just

$$p_{ind}(T_1, T_2)=p(T_1)\cdot p(T_2).$$

This situation will apply in Theorem 1. Note that in this case, for all n and m,

$$\sum_{|T_1|=n,|T_2|=m} p_{ind}(T_1, T_2) = 1.$$

On the other hand, if we consider pairs of binary trees (T_1, T_2) with the *restriction* that $|T_1|+|T_2|=n$, where n is the size of the input, the probability of the pair is

$$p(T_1, T_2) = \frac{p(T_1) \cdot p(T_2)}{1+|T_1|+|T_2|},$$

which corresponds to the model in which all the $n+1$ possible partitions of n into $n=i+j$ (being $|T_1|=i$ and $|T_2|=j$) are equally likely. This situation will apply in Theorems 2 and 3.

Note that this probability coincides with the probability of a tree formed by a root and the subtrees T_1 and T_2.

In this case we have that for all $n \geqq 0$,

$$\sum_{|T_1|+|T_2|=n} p(T_1, T_2) = 1.$$

3. Average size of the intersection of two trees. Let $\mathscr{B}$ be the set of all binary trees, and let $\bigcirc$ denote any internal node. Given trees $T_1, T_2 \in \mathscr{B}$ we wish to compute the average size of the intersection of the two trees, where the intersection of T_1 and T_2, denoted $(T_1 \cap T_2)$, is given by:

$$(T_1 \cap T_2) = \begin{cases} \langle \square \rangle & \text{if } T_1 \text{ or } T_2 \text{ is } \square, \\ \langle (T_1^l \cap T_2^l), \bigcirc, (T_1^r \cap T_2^r) \rangle & \text{otherwise.} \end{cases}$$

We shall define the size of the intersection of trees T_1 and T_2 by

$$s(T_1, T_2) = \begin{cases} 0 & \text{if } T_1 \text{ or } T_2 \text{ is } \square, \\ 1 + s(T_1^l, T_2^l) + s(T_1^r, T_2^r) & \text{otherwise.} \end{cases}$$

We wish to compute the average value of $s(T_1, T_2)$ over all the pairs (T_1, T_2) with $|T_1|+|T_2|=n$. Let $\tilde{s}(n)$ denote this average value; then we get

$$\tilde{s}(n) = \sum_{|T_1|+|T_2|=n} s(T_1, T_2) \cdot p(T_1, T_2).$$

Following the standard techniques [VF90], [GJ83], let us define the following generating function:

$$S(z) = \sum_{(T_1, T_2) \in \mathscr{B}^2} s(T_1, T_2) \cdot p(T_1, T_2) \cdot z^{|T_1|+|T_2|}.$$

We have to evaluate

$$\tilde{s}(n) = [z^n] S(z),$$

where $[z^n]S(z)$ denotes the nth coefficient in the expansion of $S(z)$. For this, let us define the generating function associated to random independence of trees T_1 and T_2:

$$S(x, y) = \sum_{(T_1, T_2) \in \mathscr{B}^2} s(T_1, T_2) p(T_1) p(T_2) x^{|T_1|} y^{|T_2|}. \tag{1}$$

It follows that

$$S(z) = \frac{1}{z} \int_0^z S(t, t)\, dt. \tag{2}$$

We use the following decomposition of the cartesian product of binary trees

$$\mathcal{B}^2 = (\square, \square) + \square \times (\mathcal{B} - \square) + (\mathcal{B} - \square) \times \square + (\mathcal{B} - \square)^2. \tag{3}$$

From (1) and using (3), we get the following partial differential equation

$$\frac{\partial^2 S(x, y)}{\partial x \partial y} = \frac{1}{(1-x)^2(1-y)^2} + \frac{2S(x, y)}{(1-x)(1-y)}, \tag{4}$$

subject to the boundary conditions: for all x and y, $S(x, 0) = 0$ and $S(0, y) = 0$. These boundary conditions are given by the intersection of a tree and a leaf and the intersection of a leaf and a tree, respectively. An equation of this type is called a hyperbolic partial differential equation (see, for example, [Cop75]).

The solution to (4) is

$$S(x, y) = \Psi(x, y) - \frac{1}{(1-x)(1-y)}, \tag{5}$$

where $\Psi(x, y)$ satisfies the homogeneous equation

$$\frac{\partial^2 \Psi}{\partial x \partial y} = \frac{2\Psi}{(1-x)(1-y)},$$

with boundary conditions $\Psi(x, 0) = \frac{1}{1-x}$ and $\Psi(0, y) = \frac{1}{1-y}$.

Making the change of variables

$$\begin{cases} X = -\sqrt{2} \ln(1-x), \\ Y = -\sqrt{2} \ln(1-y), \end{cases}$$

and setting

$$G(X, Y) = \Psi(1 - e^{-X/\sqrt{2}}, 1 - e^{-Y/\sqrt{2}}), \tag{6}$$

we finally obtain the hyperbolic differential equation

$$\frac{\partial^2 G}{\partial X \partial Y} = G, \tag{7}$$

subject to boundary conditions $G(X, 0) = e^{X/\sqrt{2}}$, $G(0, Y) = e^{Y/\sqrt{2}}$.

This system can be solved by the method of Riemann (see Appendix B) to yield

$$\begin{aligned} G(X, Y) = {} & \frac{1}{\sqrt{2}} \int_0^X e^{t/\sqrt{2}} J_0(2i\sqrt{(X-t)Y})\, dt \\ & + \frac{1}{\sqrt{2}} \int_0^Y e^{t/\sqrt{2}} J_0(2i\sqrt{(Y-t)X})\, dt + J_0(2i\sqrt{XY}), \end{aligned} \tag{8}$$

where J_0 denotes the Bessel function of the first kind of order 0.

4. Exact developments. First, we are going to get exact solutions to the size of the intersection of two trees.

From (5) and using (6) and (8), we can state the following result.

THEOREM 1. *Under the bst-model, the expected size of the intersection of two independently chosen random binary trees of sizes n and m is given by the formula*

$$[x^n y^m] S(x, y) = \frac{1}{n!m!} \sum_{j=0}^{\min(n,m)} 2^j \left(\begin{bmatrix} n \\ j \end{bmatrix} \sum_{k=j}^{m} \begin{bmatrix} m \\ k \end{bmatrix} + \begin{bmatrix} m \\ j \end{bmatrix} \sum_{k=j}^{n} \begin{bmatrix} n \\ k \end{bmatrix} - \begin{bmatrix} m \\ j \end{bmatrix} \begin{bmatrix} n \\ j \end{bmatrix} \right) - 1,$$

where, as usual, the notation $\left[{n \atop k}\right]$ *denotes the Stirling numbers of first kind* [Knu68].

On the other hand, we know by (2) that

$$[z^n]S(z) = \frac{1}{n+1} \cdot [z^n]S(z, z), \tag{9}$$

and using the previous theorem, we also get Theorem 2.

THEOREM 2. *Under the bst-model, the average size of the intersection of two trees T_1 and T_2 such that $|T_1| + |T_2| = n$ is given by the formula*

$$\tilde{s}(n) = \frac{1}{(n+1)!}\left(2 \sum_{k=0}^{n} \begin{bmatrix} n \\ k \end{bmatrix} \sum_{j=0}^{\lfloor k/2 \rfloor} \binom{k}{j} 2^j - \sum_{j=0}^{\lceil n/2 \rceil} \begin{bmatrix} n \\ 2j \end{bmatrix} \binom{2j}{j} 2^j\right) - 1.$$

5. Asymptotic results. We are interested in obtaining asymptotics to the $[z^n]S(z)$. But (9), together with (5), gives

$$\tilde{s}(n) = \frac{1}{n+1}[z^n]\Psi(z, z) - 1. \tag{10}$$

To obtain an asymptotic value for $[z^n]\Psi(z, z)$ we need the following result.

LEMMA 1.

$$[z^n]\Psi(z, z) \sim c_1 \cdot [z^n]J_0(-2\sqrt{2} \cdot i \cdot \ln(1-z)),$$

where $\sim$ stands for asymptotical equivalence and $c_1 = 3 + 2\sqrt{2}$.

The technical proof of this lemma is given in Appendix A.

The nth coefficient of $J_0(-2i\sqrt{2}\ln(1-z))$ can be evaluated applying singularity analysis. We start from [AS64, eq. 9.2.1], which states, when $|\zeta| \to \infty$ and $\arg \zeta < \pi$, that

$$J_0(\zeta) = \sqrt{\frac{2}{\pi\zeta}}\left[\cos\left(\zeta - \frac{\pi}{4}\right) + e^{|\operatorname{Im}\zeta|}O(|\zeta|^{-1})\right],$$

where $\operatorname{Im}\zeta$ denotes the imaginary part of the complex ζ.

Plugging $\zeta = -2i\sqrt{2}\ln(1-z)$ into the above equation, we obtain the following asymptotic expression when $z \to 1$:

$$J_0(-2i\sqrt{2}\ln(1-z)) = \frac{1}{\sqrt{\pi}\, 2^{5/4}} \cdot \frac{1}{\sqrt{\ln\left(\frac{1}{1-z}\right)}} \cdot \frac{1}{(1-z)^{2\sqrt{2}}}\left[1 + O\left(\frac{1}{\ln(1-z)}\right)\right].$$

And now, by standard application of transfer lemmas (see Theorem 3A in [FO90]), we have

$$[z^n]J_0(-2i\sqrt{2}\ln(1-z)) = c_2 \cdot \frac{n^{2\sqrt{2}-1}}{\sqrt{\ln n}}\left(1 + O\left(\frac{1}{\ln n}\right)\right), \tag{11}$$

where the value of the constant c_2 is given by

$$c_2 = \frac{1}{2^{5/4}\sqrt{\pi}\,\Gamma(2\sqrt{2})} = 0.1381288 \cdots.$$

Lemma 1, together with (10) and (11), gives the following result.

THEOREM 3. *Under the bst-model, the average size of the intersection of two trees behaves asymptotically as*

$$\tilde{s}(n) = c \cdot \frac{n^{2\sqrt{2}-2}}{\sqrt{\ln n}} \cdot \left(1 + O\left(\frac{1}{\ln n}\right)\right),$$

with $c = c_1 c_2 = 0.8050738 \cdots$.

Again, it should be emphasized that under the uniform model, the average size of the intersection of two trees is $1.5 \cdot (1+O(\frac{1}{n}))$, which is quite a different result from the one we just obtained.

As said in the introduction, it also follows from Theorem 3 that under the bst-distribution, the average complexity of the Tree Shuffle algorithm is also

$$2c \cdot \frac{n^{2\sqrt{2}-2}}{\sqrt{\ln n}} \cdot \left(1+O\left(\frac{1}{\ln n}\right)\right),$$

while under the uniform distribution the average complexity of the Tree Shufflle algorithm is $4(1+O(\frac{1}{n}))$ [CKS89].

6. Conclusions. It appears that the appearance of a hyperbolic partial differential equation such as the one in § 3 depends directly on the definition of probability distribution given in § 2, and it is rather independent of the nature of the problem under consideration. Current work by the authors seems to confirm this hypothesis. For instance, when considering other simple algorithms, like the equality of trees, the same methodology works and it also yields a hyperbolic differential equation. The present paper could be considered as a first treatment of the kind of framework inherent to the statistics on trees under the bst-distribution.

Appendix A: Proof of Lemma 1. Let $G(Z, Z) = A(Z) + J_0(2iZ)$ with

$$A(Z) = \sqrt{2} \int_0^Z e^{t/\sqrt{2}} J_0(2i\sqrt{(Z-t)Z})\, dt.$$

Let us recall the series expansion

$$J_0(x) = \sum_{k \geqq 0} \frac{(-1)^k}{(k!)^2} \left(\frac{x}{2}\right)^{2k};$$

then

$$A(Z) = \sqrt{2} \int_0^Z e^{t/\sqrt{2}} \left(\sum_{k \geqq 0} \frac{(-1)^k}{(k!)^2} \cdot (-(Z-t)Z)^k \right) dt$$
$$= \sqrt{2} \cdot \sum_{k \geqq 0} \frac{Z^k}{(k!)^2} \cdot \Phi_k(Z),$$

where

$$\Phi_k(Z) = \int_0^Z e^{t/\sqrt{2}} (Z-t)^k\, dt = (\sqrt{2})^{k+1} \cdot k! \cdot \sum_{j>k} \frac{1}{j!} \left(\frac{Z}{\sqrt{2}}\right)^j,$$

so we get

$$A(Z) = 2 \sum_{k \geqq 0} \frac{(Z\sqrt{2})^k}{k!} \left(\sum_{j>k} \frac{\left(\frac{Z}{\sqrt{2}}\right)^j}{j!} \right).$$

Let us consider the coefficient $a_n = [Z^n] A(Z)$. In order to evaluate the asymptotic behaviour of a_n, we shall distinguish three different cases:

- if $n = 0$, then $a_0 = 0$;
- if $n = 2p+1$, then

$$a_{2p+1} = \frac{2}{(\sqrt{2})^{2p+1}(2p+1)!} \cdot \sum_{k=0}^{p} \binom{2p+1}{k} 2^k;$$

- if $n = 2p$, then

$$a_{2p} = \frac{2}{2^p(2p)!} \cdot \sum_{k=0}^{p-1} \binom{2p}{k} 2^k.$$

It is easy to see that the value of each one of the summations involved in these expressions tends to concentrate in its last term. Therefore we can write it in the following form:

$$a_{2_p+1} = c'_p \cdot \frac{2}{(p!)^2} \cdot \frac{\sqrt{2}}{p+1},$$

$$a_{2_p} = c''_p \cdot \frac{2}{(p!)^2},$$

and it is straightforward to prove that c'_p and c''_p tend to 1 as p tends to ∞. So we conclude that $G(Z, Z)$ is asymptotically equivalent to

$$3J_0(2iZ) + \sqrt{2} \cdot \frac{d}{dZ} J_0(2iZ)$$

and we get the statement of the lemma, by making the change of variable $Z = -\sqrt{2} \ln(1-z)$.

Appendix B: Riemann's method. Riemann's method was devised to obtain solutions of linear partial differential equations of the second order (see, for instance, [Cop75, Chap. 5]). It can be applied to equations of the form

$$au_{xx} + 2hu_{xy} + bu_{yy} + 2gu_x + 2fu_y + cu = F(x, y),$$

where $u = u(x, y)$, subscripts denote partial differentiation with respect to the indicated variable(s) as usual, and a, b, h, g, f, and c are functions of x and y alone. Let the linear operator L be

$$L[u] = au_{xx} + 2hy_{xy} + bu_{yy} + 2gu_x + 2fu_y + cu.$$

Then, there exists a unique linear operator L^*, called the adjoint of L, such that $vL[u] - uL^*[v]$ is a divergence, i.e.,

$$vL[u] - uL^*[v] = \frac{\partial H}{\partial x} + \frac{\partial K}{\partial y}.$$

It can be shown, by means of Green's theorem, that

$$L^*[v] = (av)_{xx} + 2(hv)_{xy} + (bv)_{yy} - 2(gv)_x - 2(fv)_y + cv, \tag{12}$$

$$H = avu_x - u(av)_x + hvu_y - u(hv)_y + 2guv, \tag{13}$$

$$K = hvu_x - u(hv)_x + bvu_y - u(bv)_y + 2fuv. \tag{14}$$

In the particular case of hyperbolic equations, such as the one we are interested in, there is a canonical form into which any hyperbolic PDE can be transformed, using characteristic variables

$$L[u] \equiv 2u_{xy} + 2gu_x + 2fu_y + cu = F(x, y),$$

so $L^*[v] = 2v_{xy} - 2(gv)_x - 2(fv)_y + cv$, $H = vu_y - uv_y + 2gv$, and $K = vu_x - uv_x + 2fuv$.

Let C be some regular duly inclined arc for which Cauchy data is given, that is, u, u_x and u_y are known. Let the characteristics $x = x_0$ and $y = y_0$ through $P = (x_0, y_0)$

cut C at $Q=(x_1, y_0)$ and $R=(x_0, y_1)$, and let D be the domain bounded by PQ, PR, and C. Using Green's theorem and (12)-(14) it can be shown that

$$u(x_0, y_0)=\frac{1}{2}u(x_1, y_0)v(x_1, y_0; x_0, y_0)+\frac{1}{2}u(x_0, y_1)v(x_0, y_1; x_0, y_0) \tag{15}$$
$$+\frac{1}{2}\int_{QR}(K\,dx-H\,dy)+\frac{1}{2}\int\int_D v(x, y; x_0, y_0)F(x, y)\,dx\,dy,$$

where $v(x, y; x_0, y_0)$ is the so-called Riemann-Green function and must verify

$$L^*[v]=0,$$
$$v_x=fv \quad \text{on } y=y_0, \tag{16}$$
$$v_y=gv \quad \text{on } x=x_0, \quad \text{and}$$
$$v(x_0, y_0; x_0, y_0)=1.$$

We are seeking a general solution for[1] $L[G]\equiv 2G_{xy}-2G=0$, so we have $F=f=g=0$ and $c=2$, yielding $L^*=L$. On the other hand, the Cauchy data is known along the x-axis and y-axis:

$$G(x, 0)=e^{x/\sqrt{2}},$$
$$G(0, y)=e^{y/\sqrt{2}},$$
$$\left.\frac{\partial G}{\partial x}\right|_{y=0}=\frac{e^{x/\sqrt{2}}}{\sqrt{2}},$$
$$\left.\frac{\partial G}{\partial y}\right|_{x=0}=\frac{e^{y/\sqrt{2}}}{\sqrt{2}}.$$

Therefore, we take C as the arc constituted by the segments $(0, x_0)$ and $(0, y_0)$; then $Q=(0, y_0)$ and $R=(x_0, 0)$.

In order to obtain the Riemann-Green function, we try a series formula

$$v=\sum_{j\geqq 0}\frac{v_j\Gamma^j}{j!^2},$$

where v_j are functions to be determined and $\Gamma=(x-x_0)(y-y_0)$. Imposing conditions (16) we obtain

$$v(x, y; x_0, y_0)=J_0(2i\sqrt{(x-x_0)(y-y_0)}).$$

Then, applying (13) and (14) we obtain

$$K=e^{x/\sqrt{2}}\left[\frac{1}{\sqrt{2}}J_0(2i\sqrt{(x_0-x)y_0})+i\sqrt{\frac{y_0}{x_0-x}}\,J_1(2i\sqrt{(x_0-x)y_0})\right],$$
$$H=e^{y/\sqrt{2}}\left[\frac{1}{\sqrt{2}}J_0(2i\sqrt{(y_0-y)x_0})+i\sqrt{\frac{x_0}{y_0-y}}\,J_1(2i\sqrt{(y_0-y)x_0})\right],$$

[1] Multiplying the original PDE by 2, we get it in canonical form.

for $y=0$ and $x=0$, respectively. Changing dummy variables under integration signs by t, and x_0, y_0 by x, y (since the characteristics are chosen arbitrarily), (15) becomes

$$\begin{aligned} G(x,y) &= \frac{1}{2}G(x,0) + \frac{1}{2}G(0,y) + \frac{1}{2}\int_{OR} K\,dx + \frac{1}{2}\int_{OQ} H\,dy \\ &= \frac{1}{\sqrt{2}}\int_0^x e^{t/\sqrt{2}} J_0(2i\sqrt{(x-t)y})\,dt \\ &\quad + \frac{1}{\sqrt{2}}\int_0^y e^{t/\sqrt{2}} J_0(2i\sqrt{(y-t)x})\,dt + J_0(2i\sqrt{xy}). \end{aligned}$$

In the above equality, all steps are performed without any difficulty, recalling that the two appearing integrals of the type

$$-i\int e^{t/\sqrt{2}}\sqrt{\frac{\xi}{\eta-t}}\,J_1(2i\sqrt{(\eta-t)\xi})\,dt$$

can be integrated by parts, yielding

$$e^{t/\sqrt{2}} - J_0(2i\sqrt{\eta\xi}) - \frac{1}{\sqrt{2}}\int e^{t/\sqrt{2}} J_0(2i\sqrt{(\eta-t)\xi})\,dt.$$

Acknowledgment. We thank Josep Grané for pointing us to the solution of (4); Carles Simó for his advice in obtaining Lemma 1, and Philippe Flajolet for many interesting suggestions.

REFERENCES

[AS64] M. ABRAMOWITZ AND I. A. STEGUN, *Handbook of Mathematical Functions*, Dover Publications, New York, 1964.

[CDS89] R. CASAS, J. DÍAZ, AND J. M. STEYAERT, *Average-case analysis of Robinson's unification algorithm with two different variables*, Inform. Process. Lett., 31 (1989), pp. 227-232.

[CKS89] C. CHOPPY, S. KAPLAN, AND M. SORIA, *Complexity analysis of term-rewriting systems*, Theoret. Comput. Sci., 67 (1989), pp. 261-282.

[Cop75] E. T. COPSON, *Partial Differential Equations*, Cambridge University Press, Cambridge, U.K., 1975.

[Dev86] L. DEVROYE, *A note on the height of binary search trees*, J. Assoc. Comput. Mach., 33 (1986), pp. 489-498.

[FO82] P. FLAJOLET AND A. ODLYZKO, *The average height of binary trees and other simple trees*, J. Comput. System Sci., 25 (1982), pp. 171-213.

[FO90] ———, *Singularity analysis of generating functions*, SIAM J. Discrete Math., 3 (1990), pp. 216-240.

[GJ83] I. GOULDEN AND D. JACKSON, *Combinatorial Enumerations*, John Wiley, New York, 1983.

[Knu68] D. E. KNUTH, *The Art of Computer Programming: Fundamental Algorithms, Volume* 1, Addison-Wesley, Reading, MA, 1968.

[Knu73] ———, *The Art of Computer Programming: Sorting and Searching, Volume* 3, Addison-Wesley, Reading, MA, 1973.

[SF83] J. M. STEYAERT AND P. FLAJOLET, *Patterns and pattern-matching in trees: An analysis*, Inform. and Control, 58 (1983), pp. 19-58.

[VF90] J. S. VITTER AND P. FLAJOLET, *Average-case analysis of algorithms and data structures*, in Handbook of Theoretical Computer Science, Vol. A, Jan Van Leeuwen, ed., North-Holland, Amsterdam, 1990, pp. 410-440.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 33-42, February 1992

© 1992 Society for Industrial and Applied Mathematics
003

POLYNOMIAL THRESHOLD FUNCTIONS, AC^0 FUNCTIONS, AND SPECTRAL NORMS*

JEHOSHUA BRUCK† AND ROMAN SMOLENSKY‡

Abstract. This paper examines the class of polynomial threshold functions using harmonic analysis and applies the results to derive lower bounds related to AC^0 functions. A Boolean function is polynomial threshold if it can be represented as the sign of a sparse polynomial (one that consists of a polynomial number of terms). The main result of this paper is that the class of polynomial threshold functions can be characterized using their spectral representation. In particular, it is proved that an n-variable Boolean function whose L_1 spectral norm is bounded by a polynomial in n is a polynomial threshold function, while a Boolean function whose L_∞^{-1} spectral norm is not bounded by a polynomial in n is not a polynomial threshold function [J. Bruck, *SIAM J. Discrete Math.*, 3 (1990), pp. 168-177]. The motivation is that the characterization of polynomial threshold functions can be applied to obtain upper and lower bounds on the complexity of computing with networks of linear threshold elements. In this paper results related to the complexity of computing AC^0 functions are presented. More applications of the characterization theorem are presented in [J. Bruck, *SIAM J. Discrete Math.*, 3 (1990), pp. 168-177] and [K. Y. Siu and J. Bruck, *SIAM J. Discrete Math.*, 4 (1991), pp. 423-435].

Key words. Boolean functions, threshold functions, AC^0 functions, harmonic analysis, complexity

AMS(MOS) subject classifications. 68Q15, 68R99

1. Introduction.

1.1. Polynomial threshold functions.

A Boolean function $f(X)$ is a *threshold function* if

$$f(X) = sgn\,(F(X)) = \begin{cases} 1 & \text{if } F(X) > 0, \\ -1 & \text{if } F(X) < 0, \end{cases}$$

where

$$F(X) = \sum_{\alpha \in \{0,1\}^n} w_\alpha X^\alpha,$$

$$X^\alpha \stackrel{\text{def}}{=} \prod_{i=1}^{n} x_i^{\alpha_i},$$

and n is the number of variables. Throughout this paper a *Boolean function* will be defined as $f:\{1,-1\}^n \to \{1,-1\}$; namely, 0 and 1 are represented by 1 and -1, respectively. It is also assumed, without loss of generality, that $F(X) \neq 0$ or all X.

A *threshold gate* is a gate that computes a threshold function. It can be shown that any Boolean function can be computed by a single threshold gate if we allow the number of monomials in $F(X)$ to be as large as 2^n. This stimulates the following natural question: What happens when the number of monomials (terms) in $F(X)$ is bounded by a polynomial in n?

The question can be formulated by defining a new complexity class of Boolean functions.

DEFINITION. Let PT_1, for *polynomial threshold* functions, be the set of all Boolean functions that can be computed by a single threshold gate where the number of monomials is bounded by a polynomial in n.

* Received by the editors April 25, 1990; accepted for publication (in revised form) March 5, 1991.

† IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099.

‡ Computer Science Department, University of Toronto, Toronto, Canada M5S 1A4.

The main goal of this paper is to characterize PT_1 using the spectral representation of Boolean functions and to understand its relationship with the AC^0 class of Boolean functions.

1.2. Spectral representation of Boolean functions. The idea of representing Boolean functions as polynomials over the field of rational numbers was first used in the context of counting the number of equivalent Boolean functions [10]. Every Boolean function can be computed as a polynomial over the reals as follows:

$$f(X) = \sum_{\alpha \in \{0,1\}^n} a_\alpha X^\alpha.$$

This representation will be called the *polynomial representation of f.* This representation is unique and the coefficients of the polynomial $\{a_\alpha \mid \alpha \in \{0,1\}^n\}$ are called the spectral coefficients or the *spectrum* of f. The spectrum of f has many nice properties (for more details, see [10], [12]); one of them follows from the Parseval identity:

$$L_2(f) = \sum_{\alpha \in \{0,1\}^n} a_\alpha^2 = 1.$$

We are interested in the L_1 and L_∞ norms associated with the spectrum of f. Namely,

$$L_1(f) = \sum_{\alpha \in \{0,1\}^n} |a_\alpha|$$

and

$$L_\infty(f) = \max_{\alpha \in \{0,1\}^n} |a_\alpha|.$$

Note that for any Boolean function f, $L_1(f) \geqq 1$, $L_2(f) = 1$, and $L_\infty(f) \leqq 1$.

Example. Consider the function $f(x_1, x_2) = x_1 \wedge x_2$. Then

$$f(x_1, x_2) = \tfrac{1}{2}(1 + x_1 + x_2 - x_1 x_2).$$

Note that $L_1(f) = 2$, $L_2(f) = 1$, and $L_\infty(f) = \frac{1}{2}$. Recently the spectral approach proved itself to be a useful tool in the study of Boolean functions [4], [9]. Here we extend the results in [4] and establish a connection between the complexity of computing a Boolean function with threshold circuits and its spectral norms.

1.3. Some motivation. Recently, there has been considerable interest in study of the computational model of bounded depth unbounded fan-in polynomial size circuits that consist of linear threshold gates [6], [13], [15], [18]. This interest follows from recent results in complexity of circuits [8], [14], [17], which indicate that MAJORITY (hence, linear threshold functions) cannot be computed by a bounded depth unbounded fan-in polynomial size circuit that consists of $\vee$, $\wedge$, NOT, and PARITY gates. Thus, the next natural step in the analysis is adding MAJORITY as a possible gate in the computational model. Another motivation for this work comes from the area of neural networks [3], [7], where a linear threshold element is the basic processing element.

Why are we interested in *polynomial threshold functions*? It turns out that lower and upper bounds for polynomial threshold functions can be used to obtain lower and upper bounds for circuits of linear threshold functions. In particular, it was proved in [4] that the class of polynomial threshold functions is strictly contained in the class of functions that can be computed by a depth-2 circuit of linear threshold elements. For example, the upper bound obtained in this paper gives us a technique to prove upper bounds on the size of threshold circuits of depth 2.

The main tool in obtaining the results in [4] is a necessary condition for a function to be polynomial threshold: let f be a polynomial threshold function; then $L_\infty^{-1}(f)$ is bounded by a polynomial in n (number of variables).

The main result in this paper is a characterization of the functions in PT_1 using spectral norms. It can be perceived as an extension of the result in [4]. In particular, we obtain a dual result: given a Boolean function f, if $L_1(f)$ is bounded by a polynomial in n (number of variables), then it is a polynomial threshold function. Namely, we have a characterization of polynomial threshold functions using their spectral norms. We also prove that those conditions are strict, i.e., they are only necessary/sufficient conditions. Formally, let PL_1 be the class of Boolean functions for which the spectral norm L_1 is bounded by a polynomial in n and PL_∞ the class of Boolean functions for which L_∞^{-1} is bounded by a polynomial in n. Then our main result is the following theorem.

CHARACTERIZATION THEOREM.

$$PL_1 \subset PT_1 \subset PL_\infty.$$

1.4. Applications. There are two possible applications to characterization results. Sufficient conditions can be used to obtain upper bounds on the depth of a circuit that computes a certain function. For example, our characterization theorem is applied in [16] to prove the existence of depth-2, polynomial size MAJORITY circuits for comparison and for addition of two n-bit integers (recently, constructions for both functions were obtained [2]). These results also led to a construction of a depth-4, polynomial size MAJORITY circuit that computes the product of two n-bit integers.

Necessary conditions can be used to obtain lower bounds. For example, in [4] it was proved that there are functions that can be computed by a depth-2, polynomial size MAJORITY circuit but are not polynomial threshold functions.

In this paper we are mainly interested in using this approach for understanding the relation between threshold functions and AC^0 functions. In particular, it is interesting to find out whether $AC^0 \subset PL_1$. A result like this would imply that any AC^0 function can be computed with two layers of MAJORITY. We prove that this is not true, namely, that there exists an AC^0 function that has an exponential L_1 norm. Actually, we are able to prove a much stronger result. We exhibit a Boolean function such that $L_\infty^{-1} = \Omega(n^{poly\,\log(n)})$. Namely, there are AC^0 functions that cannot be computed as a sign of a sparse polynomial. This result complements that of [11] (about approximation of AC^0 functions).

From [4] we know that the class of polynomial threshold functions is strictly contained in the class of Boolean functions that can be computed by a depth-2, polynomial size circuit of MAJORITY gates. Hence, in view of this, it is natural to ask whether there are AC^0 functions that cannot be computed by a depth-2, polynomial size circuit of MAJORITY gates. We find a $\Omega(n^{poly\,\log(n)})$ lower bound on the size of a depth-2 circuit of MAJORITY gates that computes a certain AC^0 function. This provides a lower bound to the fact that three layers are sufficient to compute functions in AC^0 with MAJORITY gates [1].

The paper is organized as follows: in the next section we prove the characterization theorem, in § 3 we describe the application to AC^0 functions, and finally, we address some open problems. In the Appendix we sketch some of the related results from [4].

2. Characterizing polynomial threshold functions. In this section we characterize polynomial threshold functions using spectral norms. We will use the L_1 and L_∞ norms. Let PL_1 be the class of Boolean functions for which the spectral norm L_1 is bounded from above by a polynomial in n. And let PL_∞ be the class of Boolean functions for which L_∞^{-1} is bounded from above by a polynomial in n.

THEOREM 1.

$$PL_1 \subset PT_1 \subset PL_\infty.$$

Proof. In [4] it was proved that $PT_1 \subseteq PL_\infty$ (a sketch of the proof is included in the appendix). Hence, to prove the theorem we need to prove the following three lemmas.

1. $PL_1 \subseteq PT_1$. We do that in Lemma 1 below by using probabilistic arguments.

2. The inclusion is proper, i.e., $PL_1 \subset PT_1$, by exhibiting a function f such that $f \in PT_1$ but $f \notin PL_1$. We prove that the $\mathrm{EXACT}_{k,2k}$ function (outputs-1 if and only if exactly half of its inputs are -1) is in PT_1 (Lemma 2) but not in PL_1 (Lemma 3).

3. The inclusion $PT_1 \subseteq PL_\infty$ is proper by exhibiting a function f, such that $f \in PL_\infty$ but $f \notin PT_1$ (Lemma 4).

To prove that $PL_1 \subseteq PT_1$, we actually prove a stronger result.

LEMMA 1. *For any Boolean function $f(X)$, there is a polynomial $F(X)$, with at most $2nL_1(f)$ terms, such that $f(X) = \operatorname{sgn}(F(X))$.*

Proof. The proof is obtained by using the probabilistic method [5].

Let $\{a_\alpha \mid \alpha \in \{0, 1\}^n\}$ be the spectral coefficients of $f(X)$. That is,

$$f(X) = \sum_{\alpha \in \{0,1\}^n} a_\alpha X^\alpha.$$

We will prove that a polynomial $F(X)$, such that $f(X) = \operatorname{sgn}(F(X))$, exists by constructing it from the polynomial representation of $f(X)$.

Define a probability distribution over the α's, $\alpha \in \{0, 1\}^n$, as follows:

$$p_\alpha = \frac{|a_\alpha|}{L_1(f)}.$$

We choose the terms to be included in $F(X)$ according to the foregoing probability distribution. A term $\operatorname{sgn}(a_\alpha)X^\alpha$ is included in $F(X)$ with probability p_α. Formally, define m independently and identically distributed (i.i.d.) random variables Z_i, where $1 \leqq i \leqq m$. For all α, $Z_i = \operatorname{sgn}(a_\alpha)X^\alpha$ with probability p_α. These are the m monomials of $F(X)$. The value m will be determined later. Namely,

$$F(X) = \sum_{i=1}^{m} Z_i.$$

Hence, $F(X)$ is a polynomial constructed by selecting the monomials at random. Note that F may contain duplications of any term. For any given X, $F(X)$ is a random real variable. The first two moments of this random variable are given by:

$$\begin{aligned} E(Z_i(X)) &= \sum_\alpha p_\alpha \operatorname{sgn}(a_\alpha)X^\alpha \\ &= \sum_\alpha \frac{|a_\alpha|}{L_1(f)} \operatorname{sgn}(a_\alpha)X^\alpha \\ &= \frac{1}{L_1(f)} \sum_\alpha a_\alpha X^\alpha \\ &= \frac{f(X)}{L_1(f)}. \end{aligned}$$

So

$$E(F(X)) = \frac{mf(X)}{L_1(f)}$$

and

$$\operatorname{Var}(F(X)) = m\left(1 - \frac{1}{L_1^2(f)}\right).$$

Hence, choosing $m \geqq 2nL_1^2(f)$ and applying the *Central Limit Theorem*, for n sufficiently large,

$$\text{Prob}\,[f(X) \neq \text{sgn}\,(F(X))] \leqq \exp\left(-\frac{m}{2(L_1^2(f)-1)}\right) \leqq \exp(-n) < 2^{-n}.$$

And by the union bound we get that

$$\text{Prob}\,[f(X) \neq \text{sgn}\,(F(X)), \text{ for some } X] < 1.$$

Hence,

$$\text{Prob}\,[f(X) = \text{sgn}\,(F(X)), \text{ for all } X] > 0.$$

Thus, for any Boolean function f with a "small" $L_1(f)$ norm there exists a polynomial $\hat{F}(X)$ with at most $2nL_1^2(f)$ monomials such that $f(X) = \text{sgn}\,(\hat{F}(X))$ for all $X \in \{1, -1\}^n$. □

Remarks. 1. It follows from Lemma 1 that if $L_1(f)$ is polynomially "small", then there exist a sparse polynomial $F(X)$ such that $f(X) = \text{sgn}\,(F(X))$. Namely, $PL_1 \subseteq PT_1$.

2. Using the same argument as in Lemma 1 we can also prove the existence of approximations. Formally, given a function f, there exists a polynomial $F(X)$ with $O(L_1^2(f)n^{2k+1})$ monomials such that

$$|F(X) - f(X)| \leqq n^{-k}.$$

For more details and applications, see [16].

3. Lemma 1 applies also to depth-2 threshold circuits. From [4] we know that a threshold function with m terms can be computed by a depth-2 threshold circuit of size mn. Hence, given a function f, there exists a depth-2 circuit of linear threshold elements of size $O(L_1^2(f)n^2)$ that computes f.

Before we prove the next two lemmas we define the following function.

DEFINITION 1. $\text{EXACT}_{k,n}$ is a Boolean function of n variables that is defined as follows:

$$\text{EXACT}_{k,n}(X) = \begin{cases} -1 & \text{if there are exactly } k\ -1\text{'s in } X, \\ 1 & \text{otherwise.} \end{cases}$$

LEMMA 2.

$$\text{EXACT}_{k,2k} \in PT_1.$$

Proof. We prove this by constructing a polynomial $F(X)$ such that $\text{EXACT}_{k,2k}(X) = \text{sgn}\,(F(X))$. Let

$$F(X) = (k-1) + x_1x_2 + x_1x_3 + \cdots + x_{2k-1}x_{2k}.$$

Namely, $F(X)$ consists of a constant term $(k-1)$ and all the $\binom{2k}{2}$ products of two variables. Suppose X consists of $(k+m)$ -1's and $(k-m)$ 1's. Note that we want $F(X) < 0$ if and only if $m = 0$. Since $F(X)$ is symmetric, we may calculate the value of $F(X)$ as a function of m. The number of nonconstant terms in $F(X)$ that are -1 is exactly

$$k^2 - m^2 = (k+m)(k-m)$$

and the rest of the terms are 1. Hence, for X having m -1's,

$$\begin{aligned} F(X) &= (k-1) + \tbinom{2k}{2} - 2(k^2 - m^2) \\ &= 2m^2 - 1. \end{aligned}$$

Clearly, $\text{EXACT}_{k,2k}(X) = \text{sgn}\,(F(X))$. □

LEMMA 3.

$$L_1(\mathrm{EXACT}_{k,2k}) \geqq \frac{2^k}{k}.$$

Proof. The idea is to compute the spectral coefficients that correspond to monomials with exactly k variables. Since $\mathrm{EXACT}_{k,2k}$ is a symmetric function, the spectral coefficients that correspond to monomials with k variables are identical. Let us compute the coefficient corresponding to the term $x_1x_2x_3 \cdots x_k$, to be denoted by $\hat{a}$. We claim that

$$\hat{a} = \frac{(-1)}{2^{2k-1}} \sum_{i=0}^{k} (-1)^i \binom{k}{i}^2.$$

The proof is as follows (note that $\sum_X x_1x_2 \cdots x_k = 0$):

$$\hat{a} = \frac{1}{2^{2k}} \sum_{X \in \{-1,1\}^n} x_1x_2 \cdots x_k \, \mathrm{EXACT}_{k,2k}(X) \tag{1}$$

$$= \frac{1}{2^{2k}} \sum_{X \in \{-1,1\}^{2k}} x_1x_2 \cdots x_k (\mathrm{EXACT}_{k,2k}(X) - 1). \tag{2}$$

The term $(\mathrm{EXACT}_{k,2k}(X) - 1)$ in (2) is -2 if and only if the number of -1's in X is exactly k and zero otherwise. The sign of $x_1x_2 \cdots x_k$ is the parity of the number of ones in the first k variables and the result follows. Hence, for even k we have

$$\hat{a} = \frac{(-1)}{2^{2k-1}} \sum_{i=0}^{k} (-1)^i \binom{k}{i}^2 = \frac{-1}{2^{2k-1}} (-1)^{k/2} \binom{k}{k/2}.$$

And,

$$L_1(\mathrm{EXACT}_{k,2k}) \geqq \binom{2k}{k} \frac{1}{2^{2k-1}} \binom{k}{k/2} \tag{3}$$

$$\geqq \frac{2^k}{k}. \tag{4}$$

□

Before proving the next lemma we define the following function.

DEFINITION 2. The complete quadratic (CQ_n) function is a Boolean function of n variables such that

$$CQ_n(X) = (x_1 \wedge x_2) \oplus (x_1 \wedge x_3) \oplus \cdots \oplus (x_{n-1} \wedge x_n).$$

That is, CQ_n consists of the parity of all the $\binom{n}{2}$ AND's between pairs of variables.

LEMMA 4.

$$PT_1 \subset PL_\infty.$$

Proof. We prove that the inclusion is proper by constructing a function which is not in PT_1 but is in PL_∞. Consider the complete quadratic function ($CQ_n(X)$) which is defined in Definition 2 above. From [4] we know that $CQ_n(X)$ is not in PT_1 (and also not in PL_∞). Consider the function $f_{n+1}(X)$ of $n+1$ variables, which is constructed from $CQ_n(X)$ as follows:

$$f_{n+1}(X) = CQ_n(X) \wedge x_{n+1}.$$

The function $f_{n+1}(X)$ is not in PT_1 because $CQ_n(X)$ can be obtained from f_{n+1} by projection (setting x_{n+1} to logical 1). Next we show that $f_{n+1}(X)$ is in PL_∞ by exhibiting

a spectral coefficient that is "large"; that is, not exponentially small. We compute the spectral coefficient that corresponds to the constant term in f_{n+1}. Writing the polynomial representation of $f_{n+1}(X)$ we have,

$$f_{n+1}(X) = \tfrac{1}{2}(1 + CQ_n(x_1x_2, \cdots, x_n) + x_{n+1} - x_{n+1}CQ_n(x_1x_2, \cdots, x_n)).$$

From [4], if $CQ_n(X) = \sum_\alpha a_\alpha X^\alpha$, then for all α, $|a_\alpha| = 2^{-(n/2)}$. Hence, the spectral coefficient corresponding to the constant term is at least 0.25 and $(1/L_\infty(f_{n+1})) \leqq 4$. That is, $f_{n+1} \in PL_\infty$. □

Remark. Using the same argument as in Lemma 4 one can prove a more general result: Let $f \notin PL_\infty$, then $(f \wedge x_{n+1}) \in PL_\infty$. Hence, the class PL_∞ seems very unnatural in some sense. However, note that the motivation for this class is for proving lower bounds. Namely, if $f \in PL_\infty$, then it means that f cannot be computed as the sign of a sparse polynomial. In this sense, it is a desirable property that we can construct functions that are not in PL_∞ and also not in PT_1.

3. AC^0 functions and spectral norms. One of the possible applications of our results is to obtain bounds on the complexity of Boolean functions. In particular, the complexity of computing Boolean functions with circuits of MAJORITY gates. In this section we address a few questions related to computing AC^0 functions using MAJORITY gates. See [16] for results related to the complexity of computing arithmetic functions using MAJORITY gates.

DEFINITION 3. Let MAJ_k be the class of functions that can be computed by a depth-k, polynomial size circuit of MAJORITY gates assuming that at every gate it is possible to negate any of the inputs. This model of computation is equivalent to a polynomial size circuit of linear threshold elements of depth k, such that the weights at every gate are bounded by a polynomial (in the number of variables). This follows from the fact that we can replace an input with weight w by w copies with weight 1.

The following theorem is a summary of our results related to AC^0 functions.

THEOREM 2. 1. $AC^0 \not\subset PL_1$.
2. $AC^0 \not\subset PL_\infty$.
3. $AC^0 \not\subset MAJ_2$.

In what follows we describe the proofs for the three parts of the theorem. Clearly, the above claims are related. We give the details of the proofs for all the three claims since we use a different technique for every one of them.

Note the following facts:

1.

$$PL_1 \subset MAJ_2.$$

This follows from Theorem 1, $PL_1 \subset PT_1$, and the fact that $PT_1 \subset MAJ_2$ (see [4]).

2. By the same arguments as in Lemma 1, if the L_1 spectral norm is $O(n^{poly\log(n)})$ then the function can be computed by a depth-2, $O(n^{poly\log(n)})$ size circuit of MAJORITY gates.

Hence, it is natural to ask whether there are AC^0 functions that have an exponential L_1 norm. A negative answer to this question will result in an upper bound on the complexity of computing AC^0 functions with MAJORITY gates. Unfortunately, we prove that indeed there is a function in AC^0 that has an exponential L_1 spectral norm. This result is further evidence that three layers of MAJORITY might be needed to compute a function in AC^0 [1].

LEMMA 5.

$$AC^0 \not\subset PL_1.$$

Proof. We exhibit a function $f \in AC^0$ such that $L_1(f)$ is exponential in the number of variables. We note here that this function is a linear size CNF. First, consider the following Boolean function;

$$\hat{f}_1(X) = (x_1 \wedge x_2) \vee (x_3 \wedge x_4).$$

It can be checked that $L_1(\hat{f}_1) = 3.5$. Now consider the function f, which is the AND of n copies of $\hat{f}_1$ where each copy consists of four different variables. Namely, let

$$\hat{f}_i = (x_{4i-3} \wedge x_{4i-2}) \vee (x_{4i-1} \wedge x_{4i}).$$

Then

$$f = \bigwedge_{i=1}^{n} \hat{f}_i.$$

The polynomial representation of f can be reduced to the following form (note that a logical 1 is represented by -1):

$$f = 1 - \frac{1}{2^{n-1}} \prod_{i=1}^{n} (1 - \hat{f}_i).$$

Hence,

$$L_1(f) \geqq -1 + \frac{1}{2^{n-1}} (L_1(\hat{f}_1) - 1)^n \geqq 1.25^n.$$

The result follows since $L_1(f)$ is exponential and $f \in AC^0$. □

The next natural question is whether $AC^0 \subset PL_\infty$. Namely, is there an $f \in AC^0$ such that $L_\infty^{-1}(f)$ is not bounded by a polynomial in n? This will also show that there is an AC^0 function that is not in PT_1, and it will complement the result in [11] that AC^0 functions can be approximated by a sign of a polynomial with $O(n^{poly \log(n)})$ terms. To prove the next two lemmas we need a definition and a couple of facts.

DEFINITION 4. The inner product mod-2 function of n variables is defined as follows:

$$IP2_n(X) = (x_1 \wedge x_2) \oplus (x_3 \wedge x_4) \cdots \oplus (x_{n-1} \wedge x_n).$$

Note that $IP2_n$ is defined for even n and consists of the parity of the AND's between the $n/2$ pairs.

FACTS. 1. It is well known that the parity of $\log^d(n)$ variables can be computed in depth $2d$ with AND, OR, and NOT gates, by using a tree of XOR's of $\log n$ variables. Hence, for fixed d, $IP2_{\log^d(n)}$ is in AC^0.

2. It is also known that $L_\infty(IP2_k) = 2^{-(k/2)}$ (see [12, Chap. 14]).

LEMMA 6. *There exists a Boolean function $f \in AC^0$, such that $L_\infty^{-1}(f) = \Omega(n^{poly \log(n)})$.*

Proof. The function $IP2_{poly \log(n)} \in AC^0$ (fact 1) and $L_\infty^{-1}(IP2_{poly \log(n)}) = 2^{poly \log(n)}$ (fact 2). Now from [4] (see the Appendix) we have that the number of terms in the representation as a sign of a polynomial is $\Omega(L_\infty^{-1})$ and the result follows. □

Now, recall that in [6] it was proved that the $IP2_n$ function is not in MAJ_2. Using the same technique as in [6] yields the following lemma.

LEMMA 7. *A depth-2 circuit of* MAJORITY *gates that computes $IP2_{poly \log n}$ is of size $\Omega(n^{poly \log(n)})$.*

Since $IP2_{poly\,\log n} \in AC^0$, this result constitutes the first known lower bound to the result of Allender [1]. Namely, there are AC^0 functions that cannot be computed by a depth-2 circuit of MAJORITY gates.

4. Open problems. There are a few open problems related to the results in the paper:

1. Note that the function $IP2_{poly\,\log n}$ can actually be computed with $O(n^{poly\,\log(n)})$ terms/gates. It is possible to compute *exactly* any AC^0 function by a sign of a polynomial with $O(n^{poly\,\log(n)})$ terms? A result like this implies that any AC^0 function is computable by a depth-2, $O(n^{poly\,\log(n)})$ size circuit of MAJORITY gates. We note here that in order to prove an exponential lower bound on the number of terms we will need a different technique than the one used here. The reason is that it is possible to prove (based on [11]) that for AC^0 functions $L_\infty^{-1} = O(n^{poly\,\log(n)})$.

2. A more interesting question will be to find an AC^0 function which cannot be computed by a depth-3, polynomial size circuit of MAJORITY gates. This will give a better lower bound for the result in [1].

3. Is there some fixed d, such that any AC^0 function can be computed by a depth-d, polynomial size circuit of MAJORITY gates?

Appendix: A spectral lower bound. In [4], we made the first connection between the complexity of computing with threshold functions and harmonic analysis techniques. The result in [4] that is relevant to this paper is a lower bound on the number of terms which is expressed using the spectral coefficients. This result completes the characterization theorem. For the sake of the completeness of the presentation we sketch the proof here.

DEFINITION. Let $S \subseteq \{0, 1\}^n$; a Boolean function f is an *S-threshold* function if there exist integers that we call weights (the w_α's) such that $f(X) = \operatorname{sgn}(\sum_{\alpha\in S} w_\alpha X^\alpha)$.

THEOREM 3. *Let $f(X) = \operatorname{sgn}(\sum_{\alpha\in S} w_\alpha X^\alpha)$ be an S-threshold function and let $\{a_\alpha \mid \alpha \in \{0, 1\}^n\}$ be its spectral representation; then*

$$|S| \geqq \frac{1}{L_\infty(f)}.$$

Proof. The proof is based on the following two key lemmas that are proved in [4].

LEMMA 8. *Fix $S \subseteq \{0, 1\}^n$. Let $F(X) = \sum_{\alpha\in S} w_\alpha X^\alpha$. Let $f(X)$, $X \in \{-1, 1\}^n$, be a Boolean function with spectral representation $\{a_\alpha \mid \alpha \in \{0, 1\}^n\}$. Then*

$$f(X) = \operatorname{sgn}(F(X)) \quad \forall X \in \{1, -1\}^n$$

if and only if

$$\sum_{X\in\{1,-1\}^n} |F(X)| = 2^n \sum_{\alpha\in S} w_\alpha a_\alpha.$$

LEMMA 9. *Let $F(X) = \sum_{\alpha\in S} w_\alpha X^\alpha$. Then for all $\alpha \in S$:*

$$2^n |w_\alpha| \leqq \sum_{X\in\{1,-1\}^n} |F(X)|.$$

The proof of the theorem is as follows: By Lemmas 8 and 9 for all $\alpha \in S$:

$$|w_\alpha| \leqq \sum_{\alpha\in S} w_\alpha a_\alpha.$$

We multiply the above inequality by $|a_\alpha|$ and sum over all $\alpha \in S$:

$$\sum_{\alpha\in S} |a_\alpha||w_\alpha| \leqq \sum_{\alpha\in S} |a_\alpha| \sum_{\alpha\in S} |w_\alpha||a_\alpha|.$$

Hence,

$$\sum_{\alpha \in S} |a_\alpha| \geqq 1.$$

Note that this is a stronger statement than

$$|S| \geqq \frac{1}{L_\infty(f)}. \qquad \square$$

Note that $PT_1 \subseteq PL_\infty$ follows directly from the foregoing theorem.

Acknowledgment. We thank Noga Alon for his useful observations that greatly improved this paper and Chaim Gotsman and Sunny Siu for their comments on an early draft of this paper. We also thank the referees for their useful comments and suggestions.

REFERENCES

[1] E. ALLENDER, *A note on the power of threshold circuits*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, 1989, pp. 580–581.

[2] N. ALON AND J. BRUCK, *Explicit constructions of depth-2 majority circuits for comparison and addition*, IBM Tech. Report RJ8300, IBM Almaden Research Center, San Jose, CA, August 1991.

[3] J. BRUCK, *Computing with networks of threshold elements*, Ph.D. thesis, Electrical Engineering Department, Stanford University, Stanford, CA, March 1989.

[4] ———, *Harmonic Analysis of Polynomial Threshold Functions*, SIAM J. Discrete Math., 3 (1990), pp. 168–177.

[5] P. ERDÖS AND J. SPENCER, *Probabilistic Methods in Combinatorics*, Academic Press, New York, 1974.

[6] A. HAJNAL, W. MAASS, P. PUDLÁK, M. SZEGEDY, AND G. TURÁN, *Threshold circuits of bounded depth*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, 1987, pp. 99–110.

[7] J. J. HOPFIELD, *Neural networks and physical systems with emergent collective computational abilities*, Proc. Nat. Acad. Sci. USA, 79 (1982), pp. 2554–2558.

[8] D. S. JOHNSON, *The NP-completeness column: An ongoing guide*, J. Algorithms, 7 (1986), pp. 289–305.

[9] J. KAHN, G. KALAI, AND N. LINIAL, *The influence of variables on Boolean functions*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, 1988, pp. 68–80.

[10] R. J. LECHNER, *Harmonic analysis of switching functions*, in Recent Developments in Switching Theory, A. Mukhopadhyay, ed., Academic Press, New York, 1971.

[11] N. LINIAL, Y. MANSOUR, AND N. NISAN, *Constant depth circuits, Fourier transforms, and learnability*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, 1989, pp. 574–579.

[12] F. J. MACWILLIAMS AND N. J. A. SLOANE, *The Theory of Error-Correcting Codes*, North-Holland, New York, 1971.

[13] I. PARBERRY AND G. SCHNITGER, *Parallel computation with threshold functions*, J. Comput. System Sci., 36 (1988), pp. 278–302.

[14] A. A. RAZBOROV, *Lower bounds for the size of circuits of bounded depth with basis* $\{\wedge, \oplus\}$, Math. Notes, 41 (1987), pp. 333–338.

[15] J. H. REIF, *On threshold circuits and polynomial computation*, in Proc. Second IEEE Structure in Complexity Theory Conference, 1987, pp. 118–123.

[16] K. Y. SIU AND J. BRUCK, *On the Power of Threshold Circuits with Small Weights*, SIAM J. Discrete Math., 4 (1991), pp. 423–435.

[17] R. SMOLENSKY, *Algebraic methods in the theory of lower bounds for Boolean circuit complexity*, in Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 77–82.

[18] A. C. YAO, *Circuits and local computations*, in Proc. 21st ACM Symposium on Theory of Computing, 1989, pp. 186–196.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 43-47, February 1992
© 1992 Society for Industrial and Applied Mathematics
004

THE AVERAGE COMPLEXITY OF PARALLEL COMPARISON MERGING*

MIHÁLY GERÉB-GRAUS† AND DANNY KRIZANC‡

Abstract. An optimal lower bound on the average time required by any algorithm that merges two sorted lists on Valiant's parallel computation tree model is proven.

Key words. merging, parallel comparison tree, average case complexity

AMS(MOS) subject classifications. 68P10, 68Q10

1. Introduction.

Valiant [V] introduced the parallel computation tree (PCT) model for studying parallelism in the classical comparison problems of maximum, median, merging, and sorting. The input for each problem is a set of n elements on which a linear ordering is defined. The basic operation available to processors is the comparison of two elements. With p processors, p comparisons may be performed simultaneously in one step. Depending on which of the 2^p possible results is attained, the next set of p comparisons is chosen. The computation ends when sufficient information is discovered about the relationships of the elements to specify the solution to the given problem.

The *deterministic time complexity* of a problem in this model is the number of steps required for the worst case input or the minimum depth of a tree solving the problem, as a function of the size of the input set and the number of processors used.

The model is easily extended to allow random computations. In the randomized PCT model, at each step we introduce a probability distribution over the choice of which p comparisons are to be performed. In this case, the *randomized time complexity* of a randomized PCT is the expected number of steps required on the worst case input. The *average time complexity* is the expected number of steps required by a PCT on the uniform distribution of the inputs. It is easy to see that the deterministic complexity is greater than or equal to the randomized complexity, which in turn is greater than or equal to the randomized average complexity. Since any randomized PCT can be thought of as a probability distribution over deterministic PCTs, the randomized average complexity is equal to the deterministic average complexity. Therefore, it is sufficient to consider deterministic PCTs below.

The deterministic and average (and therefore randomized) complexities of the above problems are now fairly well understood. Valiant [V] gives matching deterministic upper and lower bounds for finding the maximum. The lower bound presented there also holds for the median. Deterministic upper bounds for the median are presented in [AKSS], [P], and [AP]. The average complexity of maximum and median finding is determined in [R] and [M]. A complete characterization of sorting is given in the papers [AKS], [AAV] (deterministic), and [AA] (average).

Valiant [V] gave a deterministic $O(\frac{n}{p}+\log(\log n/\log(1+\frac{p}{n})))$ algorithm for merging two sorted lists of length n using p processors. The matching worst case deterministic

* Received by the editors June 15, 1989; accepted for publication (in revised form) March 5, 1991.

† Department of Computer Science, Tufts University, Medford, Massachusetts 02155. This author's research was supported in part by National Science Foundation grant NSF-CCR87-04513.

‡ Department of Computer Science, University of Rochester, Rochester, New York 14627. This author's research was supported in part by National Science Foundation grant NSF-DCR86-00379 and by a National Science and Engineering Research Council of Canada postgraduate scholarship.

lower bound was given by Borodin and Hopcroft [BH]. In this paper we show that the same bound holds for the average case.

A parallel algorithm is said to achieve *optimal speedup* if its running time using p processors is proportional to the sequential running time divided by p. We found that for merging two sorted lists of length n, average optimal speedup is achievable up to the point where the number of processors is $n/\log\log n$. Adding more processors does not result in more than constant reduction in the average time until the number of processors is greater than n.

In § 3 we prove a lower bound for average time complexity of merging two sorted lists, both of length n using n processors in the PCT model. In § 4, we generalize the above result by giving lower bounds for the case where we are merging sorted lists of length n and m using p processors.

2. Preliminaries. Let $A=(a_1,\cdots,a_{s-r})$ and $B=(b_1,\cdots,b_{s+r})$ be two sorted lists to be merged and let $C=\langle A,B\rangle=(c_1,\cdots,c_{2s})$ be the resulting merged list. Partition C into $s^{7/8}$ blocks, each containing $2s^{1/8}$ elements. (Note: Throughout the paper logarithms are to the base 2 and expressions take nonnegative integral values whenever this is convenient and otherwise insubstantial.) Denote the lth block by C_l

$$(C_l=(c_{(l-1)2s^{1/8}+1},\cdots,c_{l2s^{1/8}}))\quad \text{for } l=1,\cdots,s^{7/8}.$$

Let A_l and B_l be the elements of C_l in A and in B, respectively, for $l=1,\cdots,s^{7/8}$. A block is said to be *balanced* if $||A_l|-s^{1/8}|<s^{1/12}$.

By recursively applying the partitioning described above, starting with $s=n$, we define a levelled block partitioning of the input where each level of the partitioning is a refinement of the previous level. The kth level consists of $n^{1-(1/8)^k}$ blocks, each containing $2n^{(1/8)^k}$ elements for $k=0,\cdots,c\log\log n$, where $c<\frac{1}{3}$ is an arbitrarily chosen constant.

Each step of a PCT algorithm consists of making p comparisons, where p is the number of processors available. We index the comparison of a_u and b_v by (u,v). We say the lth block $C_l=(A_l,B_l)$ is *touched* by the comparison (u,v) if and only if $a_u\in A_l$ and $b_v\in B_l$. A comparison can touch at most one block. If it does not fall within a block, and the boundaries of the blocks are revealed, it provides no information about the merging of any block. If a block is touched by one or more comparisons we will consider it as entirely merged and additional comparisons are not required for it. However, if a block is left untouched by all p comparisons, no information was gained about it during that step, and it remains a subproblem containing $2s^{1/8}$ elements to be merged independently.

3. Average case lower bound. In this section we prove the following theorem.

THEOREM 1. *The average time complexity of any* PCT *that merges two sorted lists of length n using n processors is* $\Omega(\log\log n)$.

Proof. To prove the theorem we show that the majority of kth level blocks will be left untouched by comparisons made during the kth step of a merging algorithm for $k=1,\cdots,c\log\log n$. Therefore, after $O(\log\log n)$ steps the majority of the blocks, and thus elements, will be left unmerged. We first bound the expected number of blocks merged during the kth step of the algorithm. Prior to the kth step of the algorithm we reveal the block partitioning of the input up to the $(k-1)$th level. Note that if we now consider any block from the kth level of the partitioning, the uniform distribution over the inputs conditioned by the first $k-1$ levels of the partitioning yields the uniform distribution over this block.

Let $2s = 2n^{(1/8)^{k-1}}$, the size of the $(k-1)$th level blocks. We will consider a kth level block merged during the kth step if the $(k-1)$th level block of which it is a subblock was unmerged prior to the kth step and one of the following is true:

1. It is one of the first or last $s^{3/4}$ subblocks of a $(k-1)$th level block.
2. It is unbalanced.
3. It is below the first and above the last $s^{3/4}$ subblocks and a comparison touches it.

The above accounting gives an upper bound on the number of kth level blocks (and therefore the number of elements) that are merged during the kth step. We give upper bounds on the expected number of elements merged in each case.

Case 1. At most $2s^{3/4} \cdot n^{1-(1/8)^k} \cdot s^{1/8} = n^{1-(1/8)^{k+1}}$ elements are merged in this case.

Case 2. Since the $(k-1)$th level block was unmerged it must be balanced. Deciding the boundaries of a subblock of such a balanced block is equivalent to sampling elements from a space with $N = 2s$ elements, at most $M = s + s^{2/3}$ elements of which are from list A (or B). Using bounds on the hypergeometric distribution given in [C], the probability that we select $j = s^{1/8} + s^{1/12}$ or more elements from A (or B) in $h = 2s^{1/8}$ trials is at most

$$H(M, N, h, j) \leqq \exp\left(-2\left(\frac{j}{h} - \frac{M}{N}\right)^2 h\right)$$
$$\leqq \exp(-s^{1/24}) < O(s^{-1/8}).$$

Thus the expected number of elements merged in this case is at most $n^{1-(1/8)^{k+1}} \cdot O(s^{-1/8}) \cdot s^{1/8} = O(n^{1-(1/8)^{k+1}})$ elements.

Case 3. Consider an arbitrary comparison (u, v). If it is to touch the lth block, we must have $(l-1) \cdot 2s^{1/8} < u + v \leqq l \cdot 2s^{1/8}$, which implies $l = \lceil (u+v)/2s^{1/8} \rceil$. Suppose the last element of A_l is a_{d_l}. Since $|A_l| < 2s^{1/8}$, there are at most $2s^{1/8}$ possible values for d_l. These possible boundary values are $u \leqq d_l < u + 2s^{1/8}$. Thus the probability that (u, v) touches any block is equal to the probability that it touches the lth block, which is less than or equal to the probability that $u \leqq d_l < u + 2s^{1/8}$.

For a particular r_1, the probability that $d_l = l \cdot s^{1/8} - r_1$ is

$$\frac{\binom{l \cdot 2s^{1/8}}{l \cdot s^{1/8} - r_1}\binom{2s - l \cdot 2s^{1/8}}{s - s^{1/8} - (r - r_1)}}{\binom{2s}{s - r}}.$$

For a given r the above takes its maximum when sign $(r_1) =$ sign (r) and $|r_1| < |r|$. Then it is equal to

$$\frac{\binom{l \cdot 2s^{1/8}}{l \cdot s^{1/8}}\binom{2s - l \cdot 2s^{1/8}}{s - s^{1/8}}}{\binom{2s}{s}} \times \frac{\prod_{i=1}^{r_1} \frac{(l \cdot s^{1/8}) - i}{(l \cdot s^{1/8}) + i} \prod_{i=1}^{r-r_1} \frac{(s - l \cdot s^{1/8}) - i}{(s - l \cdot s^{1/8}) + i}}{\prod_{i=1}^{r} \frac{s-i}{s+i}}.$$

For $s^{3/4} \leqq l \leqq s^{7/8} - s^{3/4}$, by Stirling approximation the first expression is $O(s^{-3/8})$. In the second expression, if we order the factors at the top and the bottom, then each factor at the bottom will be bigger than the corresponding factor at the top, and as a consequence the second expression is bounded by 1.

Therefore the probability that a comparison touches any block in the given range is $O(s^{-3/8}) \cdot 2s^{1/8} = O(s^{-1/4})$. Since there are n comparisons the expected number of elements merged in this case is at most $n \cdot O(s^{-1/4}) \cdot s^{1/8} = O(n^{1-(1/8)^{k+1}})$ elements.

Combining the three cases we have during the kth step, the expected number of elements merged is at most $O(n^{1-(1/8)^{k+1}})$. Summing over the first $c \log\log n$ steps we get that the total expected number of elements merged is $o(n)$. From this the theorem follows. □

4. Further results. To complete our investigation we look at the general case where the number of processors is p, and we are merging sorted lists of length n and m. We prove the following theorem.

THEOREM 2. *The average time complexity of any* PCT *that merges two sorted lists, one of length n, the other of length $m \geqq n$, using p processors is*

1. $$\Theta\left(\frac{n}{p}\left(\log\left(\frac{m}{n}\right)+1\right)+\log\log n\right) \quad \text{if } p \leqq 2n;$$

2. $$\Theta\left(\log\log n-\log\log\left(\frac{p}{n}\right)+\frac{\log(m/n)}{\log(p/n)}\right) \quad \text{if } 2n<p \leqq \frac{n^2}{2};$$

3. $$\Theta\left(\frac{\log(m/n)}{\log(p/n)}+1\right) \quad \text{if } p \geqq \frac{n^2}{2}.$$

Proof. Case 1. $p \leqq n$. Valiant [V] showed how to merge two sorted lists of length n with $p \leqq 2n$ processors in time $O(\frac{n}{p}+\log\log n)$. To merge lists of length n and m, first merge, using p processors, the list of length n with a list of length n consisting of every $\frac{m}{n}$th element of the second list. Now insert each of the n elements from the first list into the list of length $\frac{m}{n}$ in which it belongs. With p processors using binary search this takes at most $O(\frac{n}{p}\log(\frac{m}{n}))$ time.

The average case lower bound follows from the information-theoretic lower bound of $\Omega(\log\binom{m+n}{n})$ for the average number of comparisons required for merging in the sequential setting and from the claim below.

Case 2. $n \leqq p \leqq n^2/2$. Valiant [V] showed how to merge two sorted lists of length n with $p \geqq 2n$ processors in time $O(\log\log n-\log\log(\frac{p}{n}))$. As above, to merge lists of length n and m, first merge the list of length n with a list of length n composed of every $\frac{m}{n}$th element of the second list. Then insert each of the n elements of the first list into the list of length $\frac{m}{n}$ in which it belongs. With p processors available we can assign $\frac{p}{n}$ to each element and therefore use $\frac{p}{n}$-way search. This requires $O(\log(m/n)/\log(p/n))$ time.

There are two parts to the lower bound. The first portion is immediate from the following claim.

CLAIM. *Let us express p in form*: $p=n^{1+2^{-q}}$. *If* $m<n^{1+q2^{-q}}$, *then the average time complexity of any* PCT *that merges two sorted lists, one of length n, the other of length m, using p processors is*

$$\Omega(q)=\Omega\left(\log\log n-\log\log\left(\frac{p}{n}\right)\right).$$

Proof of Claim. As in the proof of Theorem 1, we recursively partition the input into blocks. In this case the blocks at the kth level are of size $n^{(1/8)^k}+\frac{m}{n}n^{(1/8)^k}$. Using the same proof we can show that at most $O((p/n)(m/n)n^{1-(1/8)^{k+1}})$ elements are expected to be merged on the kth step. From this we conclude that, for any $c<\frac{1}{3}$, after $c\log(\log n/\log(p/n))$ steps the expected number of elements merged is $o(m)$. This implies the average time for merging with p processors is $\Omega(q)=\Omega(\log\log n-\log\log(p/n))$. □

Further, we show that the merging is just as hard (for the average case) as performing $O(n)$ insertions of elements into lists of length $\frac{m}{n}$. Divide the list of length m into segments of length $\frac{m}{n}$. If we reveal to the algorithm which of the segments each of the n elements of the first list fall in, we are left with the problem of inserting these elements into their corresponding segments. With constant probability there are at least $\frac{n}{2}$ segments with at least one element to be inserted. To perform one such insertion with $\frac{2p}{n}$ processors requires $\Omega(\log(m/n)/\log(2p/n))$ time on average. (This follows from the optimality of p-way search with p processors. See [K].) Therefore, to perform $\frac{n}{2}$ independent insertions (note that information gathered in one insertion is of no use in another) with p processors requires $\Omega(\log(m/n)/\log(2p/n))$ time on average. This implies the average time for merging with p processors is $\Omega(\log(m/n)/\log(p/n))$.

Case 3. $p \geqq (n^2/2)$. The upper bound is the same as Case 2. With $p \geqq (n^2/2)$ processors, merging two sorted lists of length n requires constant time. The lower bound is the second part of the lower bound of Case 2. Note that merging always requires at least one step. □

As a corollary, we have the case of merging two sorted lists of length n using p processors.

COROLLARY 1. *The average time complexity of any* PCT *that merges two sorted lists of length n using p processors is*

$$\Theta\left(\frac{n}{p}+\log\left(\frac{\log n}{\log(1+p/n)}\right)\right). \qquad \square$$

Acknowledgments. We are greatly indebted to Les Valiant not only for raising the question but also for his guidance during all phases of the work. We would like to thank N. Alon and Y. Azar for pointing out extensions which led to Theorem 2, and for other helpful comments.

REFERENCES

[AA] N. ALON AND Y. AZAR, *The average complexity of deterministic and randomized parallel comparison sorting algorithms*, SIAM J. Comput., 6 (1988), pp. 1178–1192.

[AAV] Y. AZAR AND U. VISHKIN, *Tight complexity bounds for parallel comparison sorting*, SIAM J. Comput., 3 (1987), pp. 458–464.

[AKS] M. AJTAI, J. KOMLÓS, AND E. SZEMERÉDI, *Sorting in* c log n *parallel steps*, Combinatorica, 3 (1983), pp. 1–19.

[AKSS] M. AJTAI, J. KOMLÓS, W. L. STEIGER, AND E. SZEMERÉDI, *Optimal parallel selection has complexity* $O(\log\log N)$, J. Comput. System Sci., 38 (1989) pp. 125–133.

[AP] Y. AZAR AND N. PIPPENGER, *Parallel selection*, Discrete Appl. Math., 27 (1990), pp. 49–58.

[BH] A. BORODIN AND J. E. HOPCROFT, *Routing, merging and sorting on parallel models of computation*, J. Comput. System Sci., 30 (1985), pp. 130–145.

[C] V. CHVÁTAL, *The tail of the hypergeometric distribution*, Discr. Math., 25 (1979), pp. 285–287.

[K] C. P. KRUSKAL, *Searching, merging and sorting in parallel computation*, IEEE Trans. Comput., C-32, (1983) pp. 942–946.

[M] N. MEGGIDO, *Parallel algorithms for finding the maximum and the median almost surely in constant-time*, Tech. Report, Carnegie-Mellon University, Pittsburgh, PA, October 1982.

[P] N. PIPPENGER, *Sorting and selecting in rounds*, SIAM J. Comput., 16 (1987), pp. 1032–1038.

[R] R. REISCHUK, *Probabilistic parallel algorithms for sorting and selection*, SIAM J. Comput., 14 (1985), pp. 396–409.

[V] L. G. VALIANT, *Parallelism in comparison problems*, SIAM J. Comput., 4 (1975), pp. 348–355.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 48-53, February 1992

© 1992 Society for Industrial and Applied Mathematics
005

A NOTE ON THE HEIGHT OF SUFFIX TREES*

LUC DEVROYE†, WOJCIECH SZPANKOWSKI‡, AND BONITA RAIS§

Abstract. Consider a random word in which the individual symbols are drawn from a finite or infinite alphabet with symbol probabilities p_i, and let H_n be the height of the suffix tree constructed from the first n suffixes of this word. It is shown that H_n is asymptotically close to $2 \log n / \log (1/\sum_i p_i^2)$ in many respects: the difference is $O(\log \log n)$ in probability, and the ratio tends to one almost surely and in the mean.

Key words. suffix tree, height, trie hashing, analysis of algorithms, strong convergence algorithms on words

AMS(MOS) subject classifications. 68Q25, 68P05

C.R. classifications. 3.74, 5.25, 5.5

1. Introduction. *Tries* are efficient data structures that were developed and modified by Fredkin [14]; Knuth [19]; Larson [21]; Fagin, Nievergelt, Pippenger, and Strong [10]; Litwin [23], [24]; Aho, Hopcroft, and Ullman [2]; and others. Multidimensional generalizations were given in Nievergelt, Hinterberger, and Sevcik [26] and Régnier [30]. One kind of trie, the suffix tree, is of particular utility in a variety of algorithms on strings (Aho, Hopcroft, and Ullman [1]; McCreight [25]; Apostolico [3]). However, except for the results in Apostolico and Szpankowski [5], who give an upper bound on the expected height (see also Szpankowski [32]), very little is known about the expected behavior of suffix trees. Also noteworthy is a result by Blumer, Ehrenfeucht, and Haussler [6] who obtained asymptotics for the expected size of the suffix tree under an equal probability model. The difficulty arises from the interdependence between the keys, which are suffixes of one string. In this note, we study the height of the suffix tree. The results of our analysis find applications in many areas (Aho, Hopcroft, and Ullman [1]; Apostolico [3]). For example, suffix trees are used to find the longest repeated substring (Weiner [33]), to find all squares or repetitions in strings (Apostolico and Preparata [4]), to compute string statistics (Apostolico and Preparata [4]), to perform approximate string matching (Landau and Vishkin [20]; Galil and Park [15]), to compress text (Lempel and Ziv [22]; Rodeh, Pratt, and Even [29]), to analyze genetic sequences, to identify biologically significant motif patterns in DNA (Chung and Lawler [8]), to perform sequence assembly (Chung and Lawler [8]), and to detect approximate overlaps in strings (Chung and Lawler [8]). Consequences of our findings for an efficient design of algorithms are extensively discussed in Apostolico and Szpankowski [5].

* Received by the editors July 13, 1989; accepted for publication (in revised form) March 31, 1991.

† School of Computer Science, McGill University, 805 Sherbrooke Street West, Montreal, Canada H3A 2K6. This research was performed while the author was visiting Institut National de Recherche en Informatique et en Automatique, Domaine de Voluceau, Rocquencourt, B.P. 105, 78153 Le Chesnay, France. This research was sponsored by National Sciences and Engineering Research Council of Canada grant A3456 and by Fonds Le Chercheurs et d'Actions Concertées grant EQ-1678.

‡ Department of Computer Science, Purdue University, West Lafayette, Indiana 47907. This research was performed while the author was visiting Institut National de Recherche en Informatique et en Automatique, Domaine de Voluceau, Roquencourt, B.P. 105, 78153 Le Chesnay, France. This research was supported in part by North Atlantic Treaty Organization Collaborative grant 0057/89; by National Science Foundation grants NCR-8702115, CCR-8900305, and INT-8912631; by Air Force grant 90-0107; and by National Library of Medicine grant R01 LM05118.

§ Department of Computer Science, Purdue University, West Lafayette, Indiana 47907. This research was partially supported by National Science Foundation grant CCR-8900305 and Air Force grant 90-0107.

We consider an independently and identically distributed (i.i.d.) sequence $X_1, X_2, \cdots$ of integer-valued nonnegative random variables with $\mathbf{P}(X_1 = i) = p_i$ for $i = 0, 1, 2, \cdots$ and $\sum_i p_i = 1$. The X_i's should be considered as symbols in some alphabet. Together, they form a word $X = X_1X_2X_3 \cdots$. We do not assume that the alphabet is finite, but we will assume that no p_i is one, for otherwise all the symbols are identical with probability one. The *suffixes* Y_i of X are obtained by forming the sequences $Y_i = (X_iX_{i+1} \cdots)$. The *suffix* tree based upon $Y_1, \cdots, Y_n$ is the trie obtained when the Y_i's are used as words (for a definition of tries, see Knuth [19]; for a survey of recent results, see Szpankowski [31], [32]). Note, however, that we do not compress the trie as in a PATRICIA trie, i.e., no substrings are collapsed into one node.

In this note we study the *height* H_n of the suffix tree, which is given by

$$H_n = \max_{i \neq j, 1 \leqq i,j \leqq n} C_{ij},$$

where C_{ij} is the length of the longest common prefix of Y_i and Y_j, i.e., $C_{ij} = k$ if

$$(X_i, \cdots, X_{i+k-1}) = (X_j, \cdots, X_{j+k-1}) \quad \text{and} \quad X_{i+k} \neq X_{j+k}.$$

In the discussions to follow, we will use the standard notations for the L_r-metric: $\|p\|_r = (\sum_i p_i^r)^{1/r}$, where $0 < r < \infty$, and $\|p\|_\infty = \max_i p_i$. We write $f(n) \sim g(n)$ if $\lim_{n \to \infty} f(n)/g(n) = 1$, and we will reserve the symbol Q to stand for $1/\|p\|_2$.

THEOREM. *For a random suffix tree, $H_n / \log_Q n \to 1$ in probability. Also, for all $m \geqq 1$, $\mathbf{E}H_n^m \sim (\log_Q n)^m$.*

We will prove this result using only elementary probability theoretical tools, such as the second moment method. Nevertheless, we will in fact be able to show that for any $\varepsilon > 0$ and any sequence $\omega_n \uparrow \infty$,

$$\lim_{n \to \infty} \mathbf{P}(H_n > \log_Q n + \omega_n) = 0 \tag{1}$$

and

$$\lim_{n \to \infty} \mathbf{P}(H_n < \log_Q n - (1+\varepsilon) \log_Q \log n) = 0. \tag{2}$$

Thus, the variations of H_n are at most of the order of $\log \log n$. In § 4, we will show that the convergence in the theorem is in the almost sure sense as well.

It is interesting to note that the first asymptotic term ($\log_Q n$) is of the same order of magnitude as for the asymmetric trie when the words $Y_1, \cdots, Y_n$ are i.i.d. (Pittel [27], [28]; Szpankowski [32]). In [27], Pittel showed that $H_n / \log_Q n \to 1$ almost surely, and in [28], he showed that $H_n - \log_Q n = O(1)$ in probability. Other properties of the height of a trie under the independent model can be found in Yao [34]; Régnier [30]; Flajolet [11]; Devroye [9]; Pittel [27], [28]; Jacquet and Régnier [16]; and Szpankowski [32], who presents a survey of recent results. The reader is also referred to some other related papers, such as Kirschenhofer and Prodinger [18], Flajolet and Puech [12], Flajolet and Sedgewick [13], and Szpankowski [31].

2. Preliminary results. We present four simple lemmas. The first two are trivial. The third one is due to Apostolico and Szpankowski [5].

LEMMA 1.

$$\|p\|_\infty^2 \leqq \|p\|_2^2 \leqq \|p\|_\infty.$$

LEMMA 2. *For every $r \geqq 2$, $\|p\|_r \leqq \|p\|_2$.*

Proof. Let $f(x) = \{\sum_i p_i^x\}^{1/x}$ for $x > 0$. It is easy to show that the first derivative of $f(x)$ is negative for all $x > 0$, and hence f is a decreasing function. For details, see Szpankowski [32] and Karlin and Ost [17]. □

LEMMA 3. *For* $0<|i-j|=d$, *we have*

$$\mathbf{P}(C_{ij}\geqq k)=\left(\sum_s p_s^{l+2}\right)^r\left(\sum_s p_s^{l+1}\right)^{d-r},$$

where $l=\lfloor k/d\rfloor$ *and* $r=k-dl=k \bmod d$. *In particular, for* $|i-j|\geqq k$, *we have* $\mathbf{P}(C_{ij}\geqq k)=\|p\|_2^{2k}$.

LEMMA 4. *For* $0<|i-j|=d<k$, *we have* $\mathbf{P}(C_{ij}\geqq k)\leqq\|p\|_2^{k+d}$.

Proof. From Lemmas 2 and 3 we immediately obtain

$$\mathbf{P}(C_{ij}\geqq k)=\left(\sum_s p_s^{l+2}\right)^r\left(\sum_s p_s^{l+1}\right)^{d-r}\leqq\|p\|_2^{(l+2)r+(l+1)(d-r)}=\|p\|_2^{k+d}. \qquad \square$$

3. Proof of the theorem. We prove our theorem by showing two tight bounds for the height H_n. Roughly speaking, we shall show that for every $\varepsilon>0$ and large n the following holds: $\mathbf{P}(H_n>(1+\varepsilon)\cdot\log_Q n)\to 0$ as $n\to\infty$ (upper bound), and $\mathbf{P}(H_n<(1-\varepsilon)\cdot\log_Q n)\to 1$ as $n\to\infty$ (lower bound).

We start with an easier part of our proof, namely, the upper bound. Assume that $2\leqq k\leqq n-1$. We have, from Lemmas 2 and 4 and Bonferroni's inclusion-exclusion inequality for the probability of the union of events,

$$\begin{aligned}\mathbf{P}(\max_{i\neq j} C_{ij}\geqq k)&\leqq 2n\left(\sum_{d=1}^{k-1}\mathbf{P}(C_{1,1+d}\geqq k)+\sum_{d=k}^{n-1}\mathbf{P}(C_{1,1+d}\geqq k)\right)\\ &\leqq 2n\left(\sum_{d=1}^{k-1}\|p\|_2^{k+d}+\sum_{d=k}^{n-1}\|p\|_2^{2k}\right)\qquad(3)\\ &\leqq 2n\left(\frac{\|p\|_2^{k+1}}{1-\|p\|_2}+n\|p\|_2^{2k}\right).\end{aligned}$$

This tends to zero provided that $\|p\|_2<1$ (this is always true) and that $n\|p\|_2^k\to 0$ (for this, it suffices that $k=(\log n+\omega_n)/(-\log\|p\|_2)$, with $\omega_n\to\infty$). This establishes (1). Let u_+ be defined as $\max(u,0)$. Clearly, $\mathbf{E}H_n\leqq\log_Q n+\mathbf{E}(H_n-\log_Q n)_+$. We will show that the second term in this upper bound is $O(1)$. Indeed, by (3),

$$\begin{aligned}\mathbf{E}(H_n\log(1/\|p\|_2)-\log n)_+^m&=\int_0^\infty\mathbf{P}(H_n\log(1/\|p\|_2)-\log n>u^{1/m})\,du\\ &\leqq\int_0^\infty\left(\frac{2e^{-u^{1/m}}}{1-\|p\|_2}+\frac{2e^{-2u^{1/m}}}{\|p\|_2}\right)du<\infty.\end{aligned}$$

A matching lower bound is obtained by the second moment method. We will use a form due to Chung and Erdös [7], which states that for events A_i, we have

$$\mathbf{P}(\cup_i A_i)\geqq\frac{(\sum_i\mathbf{P}(A_i))^2}{\sum_i\mathbf{P}(A_i)+\sum_{i\neq j}\mathbf{P}(A_i\cap A_j)}.$$

Let S be the collection of pairs of indices (i,j) with $1\leqq i,j\leqq n$, and $|i-j|\geqq k$. Let A_{ij} be the event that $C_{ij}\geqq k$. Then

$$\mathbf{P}(\max_{i\neq j} C_{ij}\geqq k)\geqq\mathbf{P}(\cup_{(i,j)\in S}A_{ij})\geqq\frac{\mathcal{P}^2}{\mathcal{P}+\mathcal{Q}},$$

where

$$\mathcal{P} \stackrel{\text{def}}{=} \sum_{(i,j)\in S} \mathbf{P}(A_{ij})$$

and

$$\mathcal{Q} \stackrel{\text{def}}{=} \sum_{(i,j),(l,m)\in S;(i,j)\neq(l,m)} \mathbf{P}(A_{ij}\cap A_{lm}).$$

To prove our lower bound it is enough to show that the probability on the right-hand side (RHS) of the above tends to 1 for k slightly smaller than $\log_Q n$ ($k=\log_Q n-\omega_n$). First we note that when $k=o(n)$, then by Lemma 3,

$$\mathcal{P}=\sum_{(i,j)\in S} \mathbf{P}(A_{ij})=|S|\|p\|_2^{2k}\in[(n^2-(2k+1)n)\|p\|_2^{2k},\, n^2\|p\|_2^{2k}].$$

We decompose the collection of pairs of pairs of indices

$$\{((i,j),(l,m)):(i,j)\in S,(l,m)\in S,(i,j)\neq(l,m)\}$$

as follows into $I_1\cup I_2\cup I_3$: I_1 captures all members with $\min(|l-i|,|l-j|)\geqq k$ and $\min(|m-i|,|m-j|)\geqq k$. I_2 holds all members with either $\min(|l-i|,|l-j|)\geqq k$ and $\min(|m-i|,|m-j|)<k$, or $\min(|l-i|,|l-j|)<k$ and $\min(|m-i|,|m-j|)\geqq k$. Finally, I_3 collects all members with $\min(|l-i|,|l-j|)<k$ and $\min(|m-i|,|m-j|)<k$. By Lemmas 1 and 2,

$$\sum_{((i,j),(l,m))\in I_1} \mathbf{P}(A_{ij}\cap A_{lm})\leqq n^4\|p\|_2^{4k},$$

$$\sum_{((i,j),(l,m))\in I_2} \mathbf{P}(A_{ij}\cap A_{lm})\leqq 8kn^3\|p\|_2^{2k}\|p\|_\infty^{k}\leqq 8kn^3\|p\|_2^{3k},$$

$$\sum_{((i,j),(l,m))\in I_3} \mathbf{P}(A_{ij}\cap A_{lm})\leqq (4k)^2n^2\|p\|_2^{2k}.$$

Thus,

$$\mathcal{Q}\leqq n^4/Q^{4k}+8kn^3/Q^{3k}+16k^2n^2/Q^{2k}.$$

If we choose k such that $n\|p\|_2^k/k\to\infty$, then

$$\mathcal{Q}=\sum_{(i,j),(l,m)\in S;\,(i,j)\neq(l,m)} \mathbf{P}(A_{ij}\cap A_{lm})\sim n^4\|p\|_2^{4k}.$$

Because

$$\mathcal{Q}-\mathcal{P}^2\leqq 8kn^3/Q^{3k}+16k^2n^2/Q^{2k}+2(2k+1)n^3/Q^{4k}$$

and $\mathcal{P}\leqq n^2/Q^{2k}$, we have

$$\begin{aligned}\mathbf{P}(\max_{i\neq j} C_{ij}<k) &\leqq \frac{\mathcal{P}+\mathcal{Q}-\mathcal{P}^2}{\mathcal{P}+\mathcal{Q}}\\ &\leqq \frac{n^2/Q^{2k}+8kn^3/Q^{3k}+16k^2n^2/Q^{2k}+2(2k+1)n^3/Q^{4k}}{n^2/Q^{2k}+(1+o(1))n^4/Q^{4k}} \qquad (4)\\ &\sim \frac{8kQ^k}{n}.\end{aligned}$$

Collecting all these terms shows that $\mathbf{P}(H_n\geqq k)\to 1$ when $n\to\infty$. The lower bound in (2) follows by setting $k=\lfloor(\log n-(1+\varepsilon)\log\log n)/(-\log\|p\|_2)\rfloor$ for $\varepsilon>0$. Also,

$$\mathbf{E}H_n^m\geqq k^m\mathbf{P}(H_n\geqq k)\sim k^m$$

if k is chosen as indicated. This concludes the proof of the lower bound and of the theorem. □

4. Strong convergence.

PROPOSITION. *For the suffix tree, $H_n/\log_Q n \to 1$ almost surely.*

Proof. We observe that H_n is monotone ↑. Thus, if a_n is a monotone ↑ sequence, we have $H_n > a_n$ finitely often if $H_{2^i} > a_{2^{i-1}}$ finitely often in i. Similarly, $H_n < a_n$ finitely often if $H_{2^i} < a_{2^{i+1}}$ finitely often in i. By the Borel-Cantelli lemma, the proposition is proved if we can show that for all $\varepsilon > 0$,

$$\sum_{i=1}^{\infty} \mathbf{P}\{H_{2^i} > (1+\varepsilon) i \log_Q 2\} < \infty \tag{5}$$

and

$$\sum_{i=1}^{\infty} \mathbf{P}\{H_{2^i} < (1-\varepsilon) i \log_Q 2\} < \infty. \tag{6}$$

To show (5), we can use the inequality (3) with $n = 2^i$ and $k = \lceil (1+\varepsilon) i \log_Q 2 \rceil$. Note that $Q^k \geqq 2^{(1+\varepsilon)i}$. The ith term in (5) is not larger than

$$\frac{2n}{(Q-1)Q^k} + 2\left(\frac{n}{Q^k}\right)^2 \leqq \frac{2}{(Q-1)2^{\varepsilon i}} + \frac{2}{2^{2\varepsilon i}},$$

which is summable in i. Similarly, to verify (6), we use (4) with $n = 2^i$ and $k = \lfloor (1-\varepsilon) i \log_Q 2 \rfloor$. The ith term in (6) does not exceed

$$(1+o(1)) \frac{8kQ^k}{n} \leqq (8+o(1)) i (1-\varepsilon)(\log_Q 2) 2^{-\varepsilon i},$$

which is summable in i, as required. □

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1975.

[2] ———, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[3] A. APOSTOLICO, *The myriad virtues of suffix trees*, in Combinatorial Algorithms on Words, Springer-Verlag, Berlin, New York, 1985, pp. 85-96.

[4] A. APOSTOLICO AND F. P. PREPARATA, *Optimal off-line detection of repetitions in a string*, Theoret. Comput. Sci., 22 (1983), pp. 297-315.

[5] A. APOSTOLICO AND W. SZPANKOWSKI, *Self-alignments in words and their applications*, J. Algorithms, 1991.

[6] A. BLUMER, A. EHRENFEUCHT, AND D. HAUSSLER, *Average sizes of suffix trees* and DAWGs, Discrete Appl. Math., 24 (1989), pp. 37-45.

[7] K. L. CHUNG AND P. ERDÖS, *On the application of the Borel-Cantelli lemma*, Trans. Amer. Math. Soc., 72 (1952), pp. 179-186.

[8] W. CHUNG AND E. LAWLER, *Approximate string matching in sublinear expected time*, in Proc. 32nd IEEE Conference on the Foundations of Computer Science, 1990, pp. 116-124.

[9] L. DEVROYE, *A probabilistic analysis of the height of tries and of the complexity of triesort*, Acta Inform., 21 (1984), pp. 229-237.

[10] R. FAGIN, J. NIEVERGELT, N. PIPPENGER, AND H. R. STRONG, *Extendible hashing—a fast access method for dynamic files*, ACM Trans. Database Systems, 4 (1979), pp. 315-344.

[11] P. FLAJOLET, *On the performance evaluation of extendible hashing and trie search*, Acta Inform., 20 (1983), pp. 345-369.

[12] P. FLAJOLET AND C. PUECH, *Tree structure for partial match retrieval*, J. Assoc. Comput. Mach., 33 (1986), pp. 371-407.

[13] P. FLAJOLET AND R. SEDGEWICK, *Digital search trees revisited*, SIAM J. Comput., 15 (1986), pp. 748-767.

[14] E. H. FREDKIN, *Trie memory*, Comm. ACM, 3 (1960), pp. 490-500.

[15] Z. GALIL AND K. PARK, *An improved algorithm for approximate string matching*, SIAM J. Comput., 19 (1990), pp. 989-999.

[16] P. JACQUET AND M. RÉGNIER, *Trie partitioning process: Limiting distributions*, Lecture Notes in Computer Science, 214, Springer-Verlag, Berlin, 1986, pp. 196-210.

[17] S. KARLIN AND F. OST, *Some monotonicity properties of Schur powers of matrices and related inequalities*, Linear Algebra Appl., 68 (1985), pp. 47-65.

[18] P. KIRSCHENHOFER AND H. PRODINGER, *Further results on digital trees*, Theoret. Comput. Sci., 58 (1988), pp. 143-154.

[19] D. E. KNUTH, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[20] G. LANDAU AND U. VISHKIN, *Introducing efficient parallelism into approximate string matching*, in Proc. 18th Annual ACM Symposium on the Theory of Computing, 1986, pp. 220-230.

[21] P. A. LARSON, *Dynamic hashing*, BIT, 18 (1978), pp. 184-201.

[22] A. LEMPEL AND J. ZIV, *On the complexity of finite sequences*, IEEE Trans. Inform. Theory, IT-22 (1976), pp. 75-81.

[23] W. LITWIN, *Trie hashing*, in Proc. ACM-SIGMOD Conference on Management of Data, Ann Arbor, MI, 1981.

[24] ———, *Trie hashing: Further properties and performances*, in Proc. Internat. IEEE Conference on Foundations of Data Organization, Kyoto, 1985.

[25] E. M. MCCREIGHT, *A space-economical suffix tree construction algorithm*, J. Assoc. Comput. Mach., 23 (1976), pp. 262-272.

[26] J. NIEVERGELT, H. HINTERBERGER, AND K. C. SEVCIK, *The grid file: An adaptable, symmetric multikey file structure*, ACM Trans. Database Systems, 9 (1984), pp. 38-71.

[27] B. PITTEL, *Asymptotical growth of a class of random trees*, Ann. Probab., 13 (1985), pp. 414-427.

[28] ———, *Path in a random digital tree: Limiting distributions*, Adv. in Appl. Probab., 18 (1986), pp. 139-155.

[29] M. RODEH, V. PRATT, AND S. EVEN, *Linear algorithm for data compression via string matching*, J. Assoc. Comput. Mach., 28 (1981), pp. 16-24.

[30] M. RÉGNIER, *On the average height of trees in digital searching and dynamic hashing*, Inform. Process. Lett., 13 (1981), pp. 64-66.

[31] W. SZPANKOWSKI, *Some results on V-ary asymmetric tries*, J. Algorithms, 9 (1988), pp. 224-244.

[32] ———, *On the height of digital trees and related problems*, Algorithmica, 6 (1991), pp. 256-277.

[33] P. WEINER, *Linear pattern matching algorithms*, in Proc. 14th ACM Annual Symposium on Switching and Automata Theory, 1973, pp. 1-11.

[34] A. YAO, *A note on the analysis of extendible hashing*, Inform. Process. Lett., 11 (1980), pp. 84-86.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 54-58, February 1992

© 1992 Society for Industrial and Applied Mathematics
006

COMPUTING ALGEBRAIC FORMULAS USING A CONSTANT NUMBER OF REGISTERS*

MICHAEL BEN-OR† AND RICHARD CLEVE‡

Abstract. It is shown that, over an arbitrary ring, the functions computed by polynomial-size algebraic formulas are also computed by polynomial-length algebraic straight-line programs that use only three registers. This was previously known for Boolean formulas [D. A. Barrington, *J. Comput. System Sci.*, 38 (1989), pp. 150-164], which are equivalent to algebraic formulas over the ring $GF(2)$. For formulas over arbitrary rings, the result is an improvement over previous methods that require the number of registers to be logarithmic in the size of the formulas in order to obtain polynomial-length straight-line programs. Moreover, the straight-line programs that arise in these constructions have the property that they consist of statements whose actions on the registers are linear and bijective. A consequence of this is that the problem of determining the iterated product of n 3×3 matrices is complete (under P-projections) for algebraic NC^1. Also, when the ring is $GF(2)$, the programs that arise in the constructions are equivalent to bounded-width permutation branching programs.

Key words. algebraic computing, straight-line programs, complexity classes

AMS(MOS) subject classifications. 68Q15, 68Q40

1. Introduction. The first investigation of the computational power of programs whose on-line storage capacity is limited to a constant number of data items was made by Borodin, Dolev, Fich, and Paul [5] and Chandra, Furst, and Lipton [8]. These authors considered bounded-width branching programs computing functions from $\{0, 1\}^n$ to $\{0, 1\}$. Such programs are equivalent to straight-line programs that employ a constant number of $\{0, 1\}$-valued read/write registers and have read-only access to their inputs (disregarding linear differences in the lengths of programs). An advantage of working with straight-line programs is that, by allowing the inputs and registers to take values from general algebraic structures, they extend naturally to a model of computation on more general types of data.

One way of assessing the computational power of these programs is to relate it to other models of computation, such as circuits or formulas. Circuits and formulas, like straight-line programs, extend naturally from settings where the data is $\{0, 1\}$-valued to settings where the data takes values from general algebraic structures. Brent [7] proved that, if the algebraic structure is a ring, any formula of size s can be "restructured" to have depth $O(\log s)$. (Actually, Brent's result, as it is stated, applies to commutative rings, but it is easily modified to apply to general rings.)

Barrington [2] showed how to compute Boolean formulas of size s (or, equivalently, depth $O(\log s)$) by bounded-width branching programs of length polynomial in s. One interesting consequence of this result is that the MAJORITY function from $\{0, 1\}^n$ to $\{0, 1\}$ is computed by a bounded-width branching program of length polynomial in n. It was previously thought (and conjectured in [5]) that the MAJORITY function is not computable by a polynomial-length bounded-width branching program. (The conjecture was supported by some results obtained under restricted conditions.)

* Received by the editors May 22, 1990; accepted for publication (in revised form) March 21, 1991. A preliminary version of this paper was presented at the 20th Annual ACM Symposium on Theory of Computing, Chicago, Illinois, 1988 [4].

† Institute of Mathematics and Computer Science, Hebrew University, Jerusalem, Israel.

‡ Department of Computer Science, University of Calgary, Calgary, Alberta, Canada T2N 1N4. This research was conducted while this author was at the University of Toronto and partially supported by a Natural Sciences and Engineering Research Council of Canada postgraduate scholarship.

Our result is the following. Let $f(x_1, \cdots, x_n)$ be an algebraic formula of size s over an arbitrary ring $(\mathcal{R}, +, \cdot, 0, 1)$. We show how to construct a straight-line program of length polynomial in s that computes $f(x_1, \cdots, x_n)$ and uses only three registers. (Our result directly bounds the length of the straight-line program by 4^d, where d is the depth of the formula. Brent's result [7] allows us to assume that $d \in O(\log s)$.)

In the special case where the ring is $GF(2)$, our result is equivalent to Barrington's [2]. Over general rings, we obtain an improvement over the previously known (straightforward) method that involves an evaluation of each node in the formula in a "depth-first traversal" order. This latter method requires a number of registers that is equal to the depth of the formula, which, in general, is at least logarithmic in the size of the formula.

Also, the straight-line programs that arise in our constructions have a special form: they consist of statements that apply special invertible linear operations to the registers. More precisely, if one regards each possible configuration of values of the three registers as a vector in $\mathcal{R}^3$, then the effect of executing a statement of these programs is equivalent to multiplying this vector by a 3×3 matrix with determinant 1 (and one entry of this matrix is an input or its negation, and the other entries are constants). Thus, the statements that constitute these programs can be viewed as elements of $SL_3(\mathcal{R})$, the *special linear group*, consisting of 3×3 matrices with determinant 1. In Barrington's constructions [2], the statements of the programs can be viewed as elements of the group S_5, of permutations on a five-element set. To further compare our results, we note that, when the ring is $GF(2)$, our programs are (in the language of Barrington) "width-7 permutation branching programs" and, when $\mathcal{R}$ is finite, our programs are "$SL_3(\mathcal{R})$-permutation branching programs."

The main motivation for this research is to investigate alternate characterizations of the complexity class "algebraic NC^1" (functions computed by logarithmic-depth algebraic circuits). In addition, Kilian [10] has shown that the fact that the programs that arise in Barrington's constructions [2] are permutation programs has applications in the design of cryptographic protocols. By expressing Boolean formulas as bounded-width permutation branching programs, Kilian shows how to construct cryptographic protocols that perform "oblivious function evaluations" of Boolean formulas with a constant number of rounds of interaction (whereas previously proposed constructions require $\Omega(\log s)$ rounds, for general formulas of size s). Using our results, this can be extended to formulas over other rings, such as the integers (see Bar-Ilan and Beaver [1] for a particular construction).

The proof of our main result is partly motivated by the constructions used by Coppersmith and Grossman [9] and Barrington [2].

2. Models of computation. Let $(\mathcal{R}, +, \cdot, 0, 1)$ be an arbitrary ring. In this section, we define formulas and straight-line programs over $\mathcal{R}$. Our definition of formulas is very standard (as circuits that are trees). Our straight-line program model is similar to conventional models (in which there is a set of registers that contain single ring elements and statements perform single ring operations) except that our statements each perform *two* ring operations. Due to the particular form of the operations in these straight-line programs, we call them "linear bijection straight-line programs." Any statement in such a program can be easily simulated by two statements in a more conventional straight-line program if one additional register is available.

DEFINITION 1. A *formula* over $(\mathcal{R}, +, \cdot, 0, 1)$ of depth d is defined as follows. A depth 0 formula is either c, for some $c \in \mathcal{R}$ (a *constant*), or x_u, for some $u \in \{1, 2, \cdots\}$ (an *input*). For $d > 0$, a depth d formula is either $(f+g)$ or $(f \cdot g)$, where f and g are

formulas of depth d_f and d_g (respectively) and $d = \max(d_f, d_g)+1$. The *size* of a formula is defined as follows. A depth 0 formula has size 1, and if f and g have sizes s_f and s_g (respectively), then the sizes of $(f+g)$ and $(f \cdot g)$ are both $s_f + s_g + 1$. A formula computes a function from $\mathcal{R}^n$ to $\mathcal{R}$ in a natural way (where n is the number of distinct inputs occurring in the formula).

DEFINITION 2. A *linear bijection straight-line program* (LBS *program*) over $(\mathcal{R}, +, \cdot, 0, 1)$ is a sequence of assignment statements of the form

$$R_j \leftarrow R_j + (R_i \cdot c); \quad \text{or}$$
$$R_j \leftarrow R_j - (R_i \cdot c); \quad \text{or}$$
$$R_j \leftarrow R_j + (R_i \cdot x_u); \quad \text{or}$$
$$R_j \leftarrow R_j - (R_i \cdot x_u),$$

where $i, j \in \{1, \cdots, m\}$, $i \neq j$, $c \in \mathcal{R}$, and $u \in \{1, \cdots, n\}$. $R_1, \cdots, R_m$ are *registers* and $x_1, \cdots, x_n$ are *inputs* (m is the number of registers and n is the number of inputs). The *length* of an LBS program is the number of statements it contains. LBS programs compute functions from $\mathcal{R}^n$ to $\mathcal{R}$ in a natural way, provided that we have some fixed convention about the initial values of registers, and about which register's final value is taken to be the output of the computation (we will specify these later).

For each specific value of the inputs $x_1, \cdots, x_n$, each statement in an LBS program induces a transformation on the row vector consisting of the values of the registers $(R_1, \cdots, R_m)$. This transformation can be represented by the matrix whose diagonal entries are 1, whose ijth entry is $\pm c$ or $\pm x_u$ (depending on which of the four basic forms the statement takes), and whose other entries are 0. Executing a statement is equivalent to multiplying $(R_1, \cdots, R_m)$ on the right by the corresponding matrix. For example, the statement $R_1 \leftarrow R_1 + (R_2 \cdot x_1)$ corresponds to the matrix (when $m = 3$)

$$\begin{pmatrix} 1 & 0 & 0 \\ x_1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

In this manner, the statements in an LBS program correspond to elements of $SL_m(\mathcal{R})$, the special linear group consisting of $m \times m$ matrices with determinant 1. In particular, in the language of Valiant [11], an LBS program of length l that uses m registers can be viewed as a "P-projection" of an iterated product of l $m \times m$ matrices.

Also, in the language of Barrington [2], when the underlying ring is $GF(2)$, each LBS program that uses m registers is equivalent to a "permutation branching program" of width $2^m - 1$ (where the "states" are the nonzero elements of $\{0, 1\}^m$). Barrington also defines "G-permutation branching programs" for any finite group G. When $\mathcal{R}$ is finite, our LBS programs correspond to $SL_m(\mathcal{R})$-permutation branching programs.

3. Main result.

DEFINITION 3. Let $f(x_1, \cdots, x_n)$ be an arbitrary formula over $\mathcal{R}$. For distinct $i, j \in \{1, \cdots, m\}$, we say that an LBS program *offsets* R_j *by* $+R_i \cdot f(x_1, \cdots, x_n)$ if it transforms the values of the registers as follows. R_j is incremented by the value of R_i times $f(x_1, \cdots, x_n)$ and (importantly) all other registers incur no net change (i.e., for all $k \neq j$, R_k has the same final value as its initial value). For example, the single statement $R_1 \leftarrow R_1 + (R_2 \cdot x_1)$ offsets R_1 by $+R_2 \cdot x_1$. Similarly, we say that an LBS program *offsets* R_j *by* $-R_i \cdot f(x_1, \cdots, x_n)$ if it *decrements* R_j by the value of R_i times $f(x_1, \cdots, x_n)$ and causes no net change in the values of all other registers. For example, the single statement $R_1 \leftarrow R_1 - (R_2 \cdot x_1)$ offsets R_1 by $-R_2 \cdot x_1$.

Note that to compute $f(x_1, \cdots, x_n)$ it is sufficient to construct an LBS program that offsets R_j by $+R_i \cdot f(x_1, \cdots, x_n)$ if one adopts an initialization convention where R_i is initially 1 and R_j is initially 0, and one adopts the convention that the value of R_j is the output of the computation.

For convenience, we say that we have LBS programs that offset R_j by $\pm R_i \cdot f(x_1, \cdots, x_n)$ whenever we have an LBS program that offsets R_j by $+R_i \cdot f(x_1, \cdots, x_n)$ as well as an LBS program that offsets R_j by $-R_i \cdot f(x_1, \cdots, x_n)$.

THEOREM 1. *Over an arbitrary ring* $(\mathcal{R}, +, \cdot, 0, 1)$, *any formula* $f(x_1, \cdots, x_n)$ *of depth* d *is computed by an* LBS *program that uses three registers and has length at most* $(2^d)^2$.

Proof. Recursively on d, the depth of $f(x_1, \cdots, x_n)$, we construct LBS programs that use the three registers and offset R_j by $\pm R_i \cdot f(x_1, \cdots, x_n)$ (for distinct $i, j \in \{1, 2, 3\}$).

The construction is trivial when $d = 0$: the appropriate single statement offsets R_j by $\pm R_i \cdot c$, or by $\pm R_i \cdot x_k$.

Suppose that we have LBS programs that offset R_j by $\pm R_i \cdot f(x_1, \cdots, x_n)$, and that offset R_j by $\pm R_i \cdot g(x_1, \cdots, x_n)$. Then we can offset R_j by $+R_i \cdot (f+g)(x_1, \cdots, x_n)$ by the LBS program

offset R_j by $+R_i \cdot f(x_1, \cdots, x_n)$
offset R_j by $+R_i \cdot g(x_1, \cdots, x_n)$,

and we can construct a similar program that offsets R_j by $-R_i \cdot (f+g)(x_1, \cdots, x_n)$. The interesting part of our construction is that we can offset R_k by

$$+R_i \cdot (f \cdot g)(x_1, \cdots, x_n)$$

by the LBS program

offset R_k by $-R_j \cdot g(x_1, \cdots, x_n)$
offset R_j by $+R_i \cdot f(x_1, \cdots, x_n)$
offset R_k by $+R_j \cdot g(x_1, \cdots, x_n)$
offset R_j by $-R_i \cdot f(x_1, \cdots, x_n)$.

To verify that this program has the required properties, let r_i, r_j, r_k be the values of R_i, R_j, R_k (respectively) before executing the above program. Note that register R_i is not modified by any of the four stages in the program. Register R_j is incremented by $r_i \cdot f(x_1, \cdots, x_n)$ (in the second stage) and then decremented by $r_i \cdot f(x_1, \cdots, x_n)$ (in the fourth stage), and thus register R_j incurs no net change from the execution of the program. Finally, register R_k is decremented by $r_j \cdot g(x_1, \cdots, x_n)$ (in the first stage) and then incremented by

$$(r_j + r_i \cdot f(x_1, \cdots, x_n)) \cdot g(x_1, \cdots, x_n)$$

(in the third stage). Thus, the net effect of the above program is to increment the contents of register R_k by $r_i \cdot (f \cdot g)(x_1, \cdots, x_n)$, as claimed.

Also, there is a similar construction that offsets R_k by $-R_i \cdot (f \cdot g)(x_1, \cdots, x_n)$ (simply switch the "+" and "−" in the second and fourth stages).

Since the maximum recursive factor per level of depth in this construction is four, if the depth of $f(x_1, \cdots, x_n)$ is d, the resulting LBS program has length at most $4^d = (2^d)^2$. □

Combining Theorem 1 with Brent's restructuring result [7], we obtain the following corollary.

COROLLARY 2. *Over an arbitrary ring* $(\mathcal{R}, +, \cdot, 0, 1)$, *any formula* $f(x_1, \cdots, x_n)$ *of size* s *is computed by an* LBS *program that uses three registers and has length polynomial in* s.

By recalling (from § 2) the close relationship between LBS programs and iterated products of matrices, and noting that iterated products of 3×3 matrices are easily expressed as polynomial-size formulas, we obtain the following from Corollary 2.

COROLLARY 3. *Over an arbitrary ring* $(\mathcal{R}, +, \cdot, 0, 1)$, *the problem of computing the product of* n 3×3 *matrices is complete under P-projections (as defined by Valiant* [11]) *for algebraic* NC^1 *(the class of functions computed by polynomial-size formulas). Moreover, the problem remains complete if the matrices are restricted to those that have determinant* 1.

Finally, we mention that the uniform versions of Barrington's result [2], [3] also carry over to our results in a straightforward manner. For example, Corollary 3 also holds for log-time uniform NC^1 and with respect to log-time uniform P-projections.

Acknowledgments. We would like to thank an anonymous referee for making thoughtful and constructive comments. Also, the second author thanks Charles Rackoff for raising some interesting questions concerning the cryptographic security of certain block ciphers that helped inspire this work.

REFERENCES

[1] J. BAR-ILAN AND D. BEAVER, *Non-cryptographic fault-tolerant computing in constant number of rounds of interaction*, in Proc. 8th Annual ACM Symposium on Principles of Distributed Computing, 1989, pp. 201-209.

[2] D. A. BARRINGTON, *Bounded-width polynomial-size branching programs recognize exactly those languages in* NC^1, J. Comput. System Sci., 38 (1989), pp. 150-164.

[3] D. A. BARRINGTON, N. IMMERMAN, AND H. STRAUBING, *On uniformity within* NC^1, in Proc. 3rd Annual IEEE Conference on Structure in Complexity Theory, 1988, pp. 47-59.

[4] M. BEN-OR AND R. CLEVE, *Computing algebraic formulas using a constant number of registers*, in Proc. 20th Annual ACM Symposium on Theory of Computing, 1988, pp. 254-257.

[5] A. BORODIN, D. DOLEV, F. FICH, AND W. PAUL, *Bounds for width two branching programs*, SIAM J. Comput., 15 (1986), pp. 549-560.

[6] A. BORODIN AND I. MUNRO, *The Computational Complexity of Algebraic and Numeric Problems*, American Elsevier, New York, 1975.

[7] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201-206.

[8] A. CHANDRA, M. FURST, AND R. J. LIPTON, *Multiparty protocols*, in Proc. 15th Annual ACM Symposium on Theory of Computing, 1983, pp. 94-99.

[9] D. COPPERSMITH AND E. GROSSMAN, *Generators for certain alternating groups with applications to cryptography*, SIAM J. Appl. Math., 29 (1975), pp. 624-627.

[10] J. KILIAN, *Founding cryptography on oblivious transfer*, in Proc. 20th Annual ACM Symposium on Theory of Computing, 1988, pp. 20-31.

[11] L. G. VALIANT, *Completeness classes in algebra*, in Proc. 11th Annual ACM Symposium on Theory of Computing, 1979, pp. 249-261.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 59-64, February 1992

© 1992 Society for Industrial and Applied Mathematics

007

ASYMPTOTIC ANALYSIS OF AN ALGORITHM FOR BALANCED PARALLEL PROCESSOR SCHEDULING*

LI-HUI TSAI†

Abstract. This paper considers the problem of assigning n tasks, under the constraint of balancing the number of tasks processed, to two identical processors to minimize the makespan. The authors develop an algorithm assigning $\lfloor n/2 \rfloor$ tasks to one processor and $\lceil n/2 \rceil$ to another. The absolute difference between the optimal makespan and the heuristic makespan is proved to be bounded by $O(\log n/n^2)$, almost surely, when task durations are independently and uniformly distributed on $[0, 1]$. In addition, total flow time is minimized if the tasks on each processor are sequenced in nondecreasing order of their processing times.

Key words. makespan scheduling, probabilistic algorithm analysis

AMS(MOS) subject classifications. 68C, 05Q, 68Q

1. Introduction. The parallel processor scheduling problem requires the assignment of n tasks to k identical processors so as to minimize the largest task finishing time (makespan). Many studies have evaluated the worst-case performance and asymptotic probabilistic performance of several heuristic methods. (See [3] for a survey of much of the research on this problem.) The recent widespread interest in very-large-scale integrated (VLSI) chips and flexible manufacturing systems (FMS) has focused attention on incorporating additional constraints into the scheduling problem. Specifically, it is important to balance the number of tasks assigned to the various production modules. For example, in the allocation of component types to machines that manufacture VLSI chips, it is important to keep the same number of component types on each machine because of the limited number of feeder locations (Ball and Magazine [1]). Another example is the assignment of tools to machines in flexible manufacturing systems (Tsai [10]), where, because of the capacity constraints of tool magazines, the objective is to minimize the largest sum of tool processing times assigned to any given machine while keeping the numbers of tools assigned among machines balanced.

Let the *work difference* of a schedule be the difference in the total processing times and the *cardinality difference* be the difference in the number of tasks assigned to processors. A schedule's work difference is an upper bound of its difference from the optimal makespan. In a *cardinality-balanced* schedule, the number of tasks assigned to each processor is either $\lfloor n/k \rfloor$ or $\lceil n/k \rceil$. Several probabilistic analyses of heuristic algorithms have been conducted for the makespan problem without cardinality balance. Frenk and Rinnooy Kan [5] prove that the work difference achieved with the well-known LPT (largest processing time first) is at most $O(\log n/n)$, almost surely, for uniformly and exponentially distributed processing times. Karmarkar and Karp (K&K) [6] developed the Differencing Method (DM) and proved that its work difference is at most $O(n^{-c\log n})$, almost surely, when the density function is reasonably smooth. For cardinality-balanced scheduling on two processors and uniformly distributed processing times, Lueker [7] proves that the pairing differencing method (PDM), a simpler and more natural version of DM, results in an expected work difference of $\Theta(n^{-1})$.

* Received by the editors November 20, 1989; accepted for publication (in revised form) April 9, 1991.

† Department of Decision and Information Sciences, University of Florida, Gainesville, Florida 32611-2017 (TSAI@NERVM. BITNET).

Coffman, Frederickson, and Lueker [2] developed a restricted largest first (RLF) algorithm that assigns pairs of tasks in nondecreasing order of the larger processing time of a pair. The expected work difference of RLF is equal to $\Theta(1/n)$. This paper presents an algorithm, called restricted largest difference (RLD), for two-processor scheduling with cardinality balance. The work difference of RLD is equal to $O(\log n/n^2)$, almost surely, which is an improvement over LPT, PDM, and RLF in the asymptotic regime. Like RLF, RLD also minimizes total flow time when tasks assigned to each processor are sequenced in nondecreasing order of their processing times—called SPT (shortest processing time) order in the remainder of this paper.

The RLD algorithm is described in § 2. In § 3 we prove that the work difference achieved by RLD is bounded by $O(\log n/n^2)$, almost surely, and that by reordering tasks on each processor in SPT order the minimal flow time is achieved.

2. An algorithm for systems with two processors. In the RLD algorithm, tasks are sorted in nonincreasing order of their processing times and then consecutive tasks are paired up. A dummy task with zero processing time is added if the number of tasks is odd. The pairs of tasks are sorted in nonincreasing order of their differences in processing times and are then assigned, one pair at a time, in that order. The larger task of a pair is assigned to the processor with less cumulative processing time and the smaller task is assigned to the other.

Let W_{i-1}^1 and W_{i-1}^2 denote the cumulative processing times on two processors *before* the ith pair of tasks, with processing times $\{x_i^1, x_i^2\}$, is assigned. Without loss of generality, we may assume $W_{i-1}^1 \leqq W_{i-1}^2$ and $x_i^1 \leqq x_i^2$; then x_i^2 is assigned to the first processor and x_i^1 to the second.

Let $d_{i-1} = W_{i-1}^2 - W_{i-1}^1$, $d_0 = W_0^2 = W_0^1 = 0$, and $\delta_i = x_i^2 - x_i^1$. The following equation verifies that the difference between cumulative processing times after the assignment of a pair of tasks is equal to the difference between d_{i-1} and δ_i:

$$(1) \qquad \begin{aligned} |(W_{i-1}^2 + x_i^1) - (W_{i-1}^1 + x_i^2)| &= |(W_{i-1}^2 - W_{i-1}^1) - (x_i^2 - x_i^1)| \\ &= |d_{i-1} - \delta_i|. \end{aligned}$$

When n is an even number the algorithm will assign $n/2$ tasks to each processor. When n is an odd number there will be $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ tasks assigned to the two processors. The algorithm is as follows.

ALGORITHM RLD

Step 1. Sort the tasks' processing times such that $X = \{x_{(1)}, x_{(2)}, \cdots, x_{(n)}\}$, where $x_{(1)} \geqq x_{(2)} \geqq \cdots \geqq x_{(n)}$.

Step 2. If n is an odd number, add a dummy task $x_{(n+1)}$ with processing time equal to zero.

Step 3. Form pairs of tasks from the largest to the smallest. Each pair of tasks consists of a couplet $\{x_{(2i-1)}, x_{(2i)}\}$, $i = 1, 2, \cdots, \lceil n/2 \rceil$.

Step 4. Assign pairs of tasks in nonincreasing order of their differences. Let $\delta_i = x_{(2i-1)} - x_{(2i)}$, $i = 1, 2, \cdots, \lceil n/2 \rceil$.

(1) Sort δ_i in nonincreasing order. Let σ_j be the index of the jth largest δ_i's; that is,

$$\delta_{\sigma_1} \geqq \delta_{\sigma_2} \geqq \cdots \geqq \delta_{\sigma_{\lceil n/2 \rceil}}.$$

(2) For $i = 1, 2, \cdots, \lceil n/2 \rceil$, assign $\{x_{(2\sigma_i - 1)}, x_{(2\sigma_i)}\}$ to different processors. Assign $x_{(2\sigma_i - 1)}$ to the processor with the smaller cumulative processing time and $x_{(2\sigma_i)}$ to the other.

The following example illustrates the algorithm for the case of $n=7$ tasks with processing times 20, 16, 14, 7, 6, 4, and 3.

Since n is an odd number, a dummy task $x_{(8)}$ with processing time equal to zero is added. The pairs formed in Step 3 are:

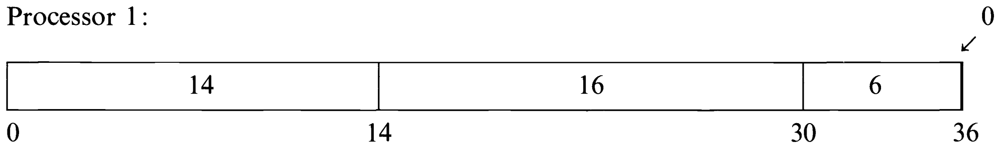
$$\{20, 16\}, \{14, 7\}, \{6, 4\}, \{3, 0\}.$$

Hence, $\delta_1=4$, $\delta_2=7$, $\delta_3=2$, $\delta_4=3$, and $\sigma_1=2$, $\sigma_2=1$, $\sigma_3=4$, $\sigma_4=3$. Since $\sigma_2<\sigma_1<\sigma_4<\sigma_3$, pairs of tasks are assigned to the two processors in the following order:

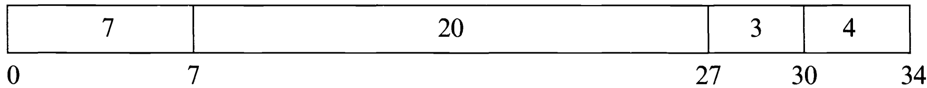
$$\{14, 7\}, \{20, 16\}, \{3, 0\}, \{6, 4\}.$$

The larger task in each pair is assigned to the processor with the smaller cumulative processing time. Suppose 14 is assigned to the first processor; then 7 will be assigned to the second. Since $7<14$, 20 will now be assigned to the second processor and 16 to the first; this continues, with 4 being the last assignment, as shown in the following Gantt charts.

Processor 1:

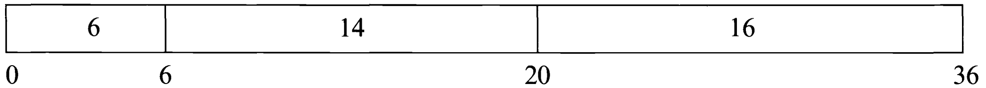


Processor 2:

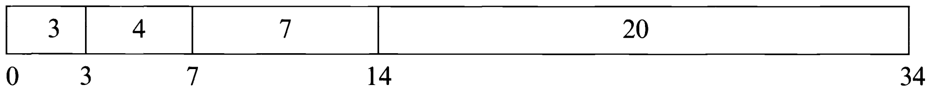


To minimize the total flow time, tasks on each processor are rearranged in SPT order. The Gantt chart below shows the sequence on each processor:

Processor 1:



Processor 2:



The total flow time of the seven tasks is $6+20+36+3+7+14+34=120$.

3. Proof of asymptotic error bound. The asymptotic analysis of RLD gives an estimate of the convergence rate of work difference. First, we will show that the work difference between two processors is at least twice the difference of the optimal and the heuristic makespans. Second, we show that the work difference is bounded by $O(\log n/n^2)$, almost surely, with uniformly distributed processing times. We use the following notation:

$M(X)$: the makespan of the heuristic algorithm for input data X.
$m(X)$: the smaller cumulative processing time for input data X.
$D(X)$: the difference between the larger and smaller cumulative processing times for input data X, i.e., $D(X)=M(X)-m(X)$.

OPT(X): the minimum makespan for input data X.

We show that the difference between $M(X)$ and OPT(X) is less than $\frac{1}{2}D(X)$, as follows:

$$
\begin{aligned}
M(X)-\mathrm{OPT}(X) &\leqq M(X)-\sum_{i=1}^{n} x_i/2 = M(x)-\frac{1}{2}[M(X)+m(X)] \\
&= \frac{1}{2}[M(X)-m(X)] = \frac{1}{2}D(X).
\end{aligned} \tag{2}
$$

Applying the above inequality to the example in § 2, we see that the difference between the optimal and heuristic makespans is at most $\frac{1}{2}(36\text{–}34)=1$.

To show that the difference between the optimal and heuristic makespans is bounded by $O(\log n/n^2)$, almost surely, we prove that $D(X)$ is bounded by $O(\log n/n^2)$, almost surely, when the processing times are independently and identically distributed (i.i.d.) random variables from the uniform distribution on $[0, 1]$. In Step 3 of RLD, δ_i's are the differences between successive order statistics of a sample of n i.i.d. uniform random variables. The following lemma, derived from Pyke [8], shows that each δ_i is distributed as the ratio of an exponential random variable to the sum of $n+1$ i.i.d. exponential variables.

LEMMA 1 (δ_i's as uniform spacing). *Let $y_1, y_2, \cdots, y_{n+1}$ be i.i.d. exponential random variables with mean* 1. *Let $S=y_1+y_2+\cdots+y_{n+1}$; then δ_i is distributed as y_{2i}/S.*

Proof. In [8, § 4.1], it is proved that $1-x_{(1)}, x_{(1)}-x_{(2)}, x_{(2)}-x_{(3)}, \cdots, x_{(n)}-0$ are distributed as $y_1/S, y_2/S, y_3/S, \cdots, y_{n+1}/S$. Since $\delta_i = x_{(2i-1)}-x_{(2i)}$, δ_i is distributed as y_{2i}/S. □

The work difference, obtained by implementing the RLD algorithm using the $x_{(i)}$'s as inputs, behaves like the difference between cumulative processing times obtained by implementing the LPT algorithm using the δ_i's as inputs. Let $D_{\mathrm{RLD}}(X)$ and $D_{\mathrm{LPT}}(X)$ be the resulting work differences for input data X, as determined by the RLD and LPT algorithms, respectively.

LEMMA 2. $D_{\mathrm{RLD}}(X)=D_{\mathrm{LPT}}(\delta)$.

Proof. The LPT algorithm will assign δ_i's in nonincreasing order of their weights, i.e., in the order $\delta_{\sigma_1}, \delta_{\sigma_2}, \cdots, \delta_{\sigma_{\lceil n/2 \rceil}}$. Let d_i be the difference of cumulative processing times on two processors *after* δ_{σ_i} is assigned. Since δ_{σ_i} is assigned to the processor with the smaller cumulative processing time, $d_i=|d_{i-1}-\delta_{\sigma_i}|$, $i>1$, and $d_1=\delta_{\sigma_1}$. Then from (1), we can verify that d_i is also the difference of cumulative processing times after $\{x_{(2\sigma_i-1)}, x_{(2\sigma_i)}\}$ are assigned to different processors by applying algorithm RLD on the x_i's. Hence, $D_{\mathrm{LPT}}(\delta)=d_{\lceil n/2 \rceil}=D_{\mathrm{RLD}}(X)$. □

Let $D_{\mathrm{LPT}}(Y)$ be the work difference achieved by the LPT algorithm with input data $y_1, y_2, \cdots, y_{\lceil n/2 \rceil}$. By Theorem 3 of Frenk and Rinnooy Kan [5], if the processing times are n i.i.d. exponential random variables, then the difference between the largest and average processing times is bounded by $O(\log n/n)$, almost surely. In (2), we showed that for systems with two processors, the work difference is twice the difference between the largest and average total processing times. Hence, $D_{\mathrm{LPT}}(Y)$ is bounded by $O(\log n/n)$, almost surely.

THEOREM 1. $D_{\mathrm{RLD}}(X)=O(\log n/n^2)$, *almost surely.*

Proof. By Lemma 2, $D_{\mathrm{RLD}}(X)=D_{\mathrm{LPT}}(\delta)$, and by Lemma 1, $D_{\mathrm{LPT}}(\delta)$ is distributed as $D_{\mathrm{LPT}}(Y)/\sum_{i=1}^{n+1} y_i$. Hence $D_{\mathrm{RLD}}(X)$ is distributed as $D_{\mathrm{LPT}}(Y)/\sum_{i=1}^{n+1} y_i$. The strong law of large numbers shows that

$$
\lim_{n\to\infty} \sum_{i=1}^{n+1} y_i/(n+1) = 1, \quad \text{a.s.} \tag{3}
$$

From Theorem 3 of [5],

$$\limsup_{n\to\infty} (n/\log n) D_{\mathrm{LPT}}(Y) < \infty, \quad \text{a.s.} \tag{4}$$

Applying Theorem 4.4. of Rudin[1] [9] to (3) and (4),

$$\limsup_{n\to\infty} (n/\log n) D_{\mathrm{LPT}}(Y) \Big/ \left[\sum_{i=1}^{n+1} y_i/(n+1) \right] < \infty, \quad \text{a.s.}$$

Therefore,

$$\lim_{n\to\infty} D_{\mathrm{LPT}}(Y) \Big/ \sum_{i=1}^{n+1} y_i = O(\log n / n^2), \quad \text{a.s.}$$

Finally, $\lim_{n\to\infty} D_{\mathrm{RLD}}(X) = O(\log n/n^2)$, almost surely. □

To reduce the total flow time, we could execute the tasks on each processor in SPT order. This reordering will not alter the work and cardinality differences, but it will achieve the advantage of minimizing the total flow time.

THEOREM 2. *After tasks are assigned according to the* RLD *algorithm, the total flow time is minimized if tasks on each processor are executed in* SPT *order.*

Proof. We show that the total flow time achieved by RLD followed by SPT is equivalent to that achieved by another optimal algorithm (called CMM) for total flow time, which was developed by Conway, Maxwell, and Miller [4]. The CMM algorithm minimizes total flow time of n tasks on k identical processors. Though cardinality balance is not an objective of CMM, it is achieved in the schedule derived by CMM. For two processors, CMM first assigns each pair of tasks $\{x_{(2i-1)}, x_{(2i)}\}$, $i = 1, 2, \cdots, \lceil n/2 \rceil$, to different processors. Tasks on each processor are then executed in SPT order. Total flow time can be expressed as a weighted sum of processing times. The weight of a task's processing time is equal to one plus the number of tasks that are executed after it on the same processor. If tasks are executed in SPT order, the weight is also equal to one plus the number of tasks with larger processing times that are assigned to the same processor. Therefore, for both CMM and RLD followed by SPT, i is the weight of $x_{(2i-1)}$ and $x_{(2i)}$, $i = 1, 2, \cdots, \lceil n/2 \rceil$. Hence they have the same total flow time. □

REFERENCES

[1] M. O. BALL AND M. J. MAGAZINE, *Sequencing of insertions in printed circuit board assembly*, Oper. Res., 36 (1986), pp. 192-201.

[2] E. G. COFFMAN, JR., G. N. FREDERICKSON, AND G. S. LUEKER, *A note on expected makespans for largest-first sequences of independent tasks on two processors*, Math. Oper. Res., 9 (1984), pp. 260-266.

[3] E. G. COFFMAN, JR., G. S. LUEKER, AND A. H. G. RINNOOY KAN, *Asymptotic methods in the probabilistic analysis of sequencing and packing heuristics*, Management Sci., 34 (1988), pp. 266-290.

[4] R. W. CONWAY, JR., W. L. MAXWELL, AND L. W. MILLER, *Theory of Scheduling*, Addison-Wesley, Reading, MA, 1967.

[5] J. B. G. FRENK AND A. H. G. RINNOOY KAN, *The asymptotic optimality of the LPT rule*, Math. Oper. Res., 12 (1987), pp. 241-254.

[6] N. KARMARKAR AND RICHARD M. KARP, *The differencing method for set partitioning*, Report, Computer Science Division, University of California, Berkeley, CA, 1982.

[1] In brief, this theorem states that the limit of the ratio of two functions equals the ratio of their limits.

[7] G. S. LUEKER, *A note on the average-case behavior of a simple differencing method for partitioning*, Oper. Res. Lett., 6 (1987), pp. 285–288.

[8] R. PYKE, *Spacings*, J. Roy. Statist. Soc. Ser. B., 7 (1965), pp. 395–449.

[9] W. RUDIN, *Principles of Mathematical Analysis*, Third Edition, McGraw-Hill, New York, 1976.

[10] L. TSAI, *The loading and scheduling problems in flexible manufacturing systems*, Ph.D. Thesis, School of Business Administration, University of California, Berkeley, CA, 1987.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 65–84, February 1992

© 1992 Society for Industrial and Applied Mathematics
008

CONDITIONS FOR UNIQUE GRAPH REALIZATIONS*

BRUCE HENDRICKSON†

Abstract. The *graph realization problem* is that of computing the relative locations of a set of vertices placed in Euclidean space, relying only upon some set of inter-vertex distance measurements. This paper is concerned with the closely related problem of determining whether or not a graph has a unique realization. Both these problems are NP-hard, but the proofs rely upon special combinations of edge lengths. If one assumes the vertex locations are unrelated, then the uniqueness question can be approached from a purely graph theoretic angle that ignores edge lengths. This paper identifies three necessary graph theoretic conditions for a graph to have a unique realization in any dimension. Efficient sequential and NC algorithms are presented for each condition, although these algorithms have very different flavors in different dimensions.

Key words. graph embeddings, graph realizations, graph algorithms, rigid graphs, connectivity

AMS(MOS) subject classifications. 05C10, 05C85

1. Introduction.

Consider a graph $G = (V, E)$ consisting of a set of n vertices and m edges, along with a real number associated with each edge. Now try to assign coordinates to each vertex so that the Euclidean distance between any two adjacent vertices is equal to the number associated with that edge. This is the *graph realization problem.* It appears in situations where one needs to know the locations of various objects, but can only measure the distances between pairs of them. Surveying and satellite ranging are among the more obvious problems that can be expressed in this form [27], [39]. A less obvious but potentially more important application has to do with determining molecular conformations. It is possible to analyze the nuclear magnetic resonance spectra of a molecule to obtain pairwise interatomic distance information [13]. Solving the graph realization problem in this context would allow us to determine the three-dimensional shape of the molecule, which is important in understanding the molecule's properties.

Unfortunately, the graph realization problem is known to be difficult. Saxe has shown it to be strongly NP-complete in one dimension and strongly NP-hard for higher dimensions [35]. In practice, this means that we are unlikely to find an efficient general algorithm to solve it. However, the graphs and edge lengths that Saxe uses in his proofs are very special and are highly unlikely to occur in practical problems.

This paper will address a closely related problem: when does the graph realization problem have a unique solution? (For our purposes, translations, rotations, and reflections of the entire space are not considered to be different realizations.) Clearly, if the location of a satellite or an atom is to be determined unambiguously the solution to the realization problem must be unique.

Saxe has shown this uniqueness problem to be as hard as the original realization problem, but again his proofs rely on very special graphs. In particular, he needs special combinations of edge lengths, implying specific algebraic relations among the coordinates of the vertices. This paper will address the more typical behavior of graphs.

* Received by the editors October 1, 1990; accepted for publication (in revised form) March 6, 1991. This research was performed while the author was at Cornell University and supported by a fellowship from the Fannie and John Hertz Foundation.

† Mathematics and Computational Science Department, Sandia National Laboratories, Albuquerque, New Mexico 87185.

A *realization* of a graph G is a function p that maps the vertices of G to points in Euclidean space. The combination of a graph and a realization is called a *framework* and is denoted by $p(G)$. A realization is *satisfying* if all the pairwise distance constraints are satisfied. Consider a set $\mathcal{S}$ with nonzero measure. A subset $\mathcal{T}$ of $\mathcal{S}$ is said to contain *almost all* of $\mathcal{S}$ if the complement of $\mathcal{T}$, $\{q \in \mathcal{S} | q \notin \mathcal{T}\}$, has measure zero. A realization is said to be *generic* if the vertex coordinates are algebraically independent over the rationals. This computationally unrealistic requirement is actually stronger than we truly need. We just have to avoid several specific algebraic dependencies. However, the set of generic realizations is dense in the space of all realizations, and almost all realizations are generic.

Restricting ourselves to generic realizations will greatly simplify our analysis. It will allow us to ignore the edge distances and base our analysis solely on the underlying graph. The results we develop will apply to graphs in almost all realizations. However, nongeneric realizations might have different properties.

How can a framework have multiple realizations? There are several distinct manners in which nonuniqueness can appear. First, the framework can be susceptible to continuous deformations, like the one in Fig. 1. The rightmost vertex in this graph

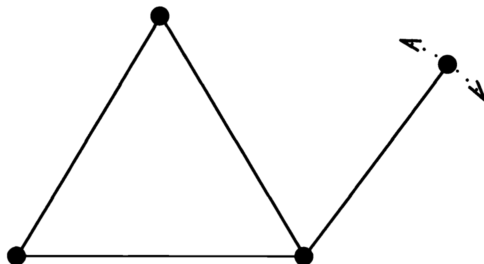


FIG. 1. *A flexible framework in two dimensions.*

can pivot freely since it is underconstrained. A framework that can be continuously deformed while still satisfying all the constraints is said to be *flexible*; otherwise it is *rigid*. Even a rigid framework can suffer from nonuniqueness. The rigid framework in Fig. 2 has two realizations in the plane. One half of the graph can reflect across the

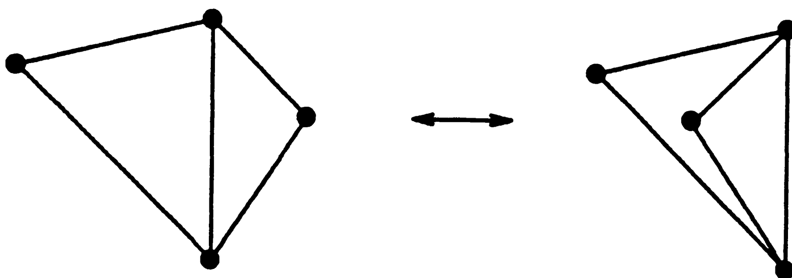


FIG. 2. *A graph with two realizations in the plane.*

central two vertices. Continuous deformations and graph rigidity will be discussed in §2. Although graph rigidity is a well-studied problem, the connections to the graph realization problem have not been well explored. Discontinuous transformations like

reflections will be covered in §§3 and 4. The definition of redundant rigidity in §4 and its importance in this context is entirely new.

The graph realization problem can be posed in any dimension. Clearly, the most practically interesting dimensions are two and three. Where possible, this paper will present the most general results. However, there are some substantial theoretical differences between two-space and three-space that will be elucidated in §2. These will lead to completely different algorithms for these different dimensions.

2. Graph rigidity.

A graph that has a unique realization cannot be susceptible to continuous flexings. It must be rigid. Questions about the rigidity of graphs have occupied mathematicians for centuries. More recently, structural engineers have been drawn to the problem because of novel building architectures like geodesic domes. The framework of a building can be thought of as a set of rigid rods, joined at their endpoints. One can consider the endpoints to be vertices of a graph and the rods to be edges of a fixed length. For the building to bear weight, the corresponding framework must be rigid. For an old problem with an easy description, the characterization of rigid graphs has proved to be difficult, and many important questions remain unanswered.

Section 2.1 will develop the essentials of rigidity theory, stressing the importance of the rigidity matrix. A more complete discussion can be found in some of the references [2], [3], [33], [11]. Section 2.2 will present sequential and parallel algorithms for rigidity testing.

2.1. Basic concepts.

A mathematical analysis of rigidity requires a formal definition of our intuitive notion of a flexible framework. Everything in this section occurs in an arbitrary Euclidean dimension d.

A *finite flexing* of a framework $p(G)$ is a family of realizations of G, parameterized by t so that the location of each vertex i is a differentiable function of t and $(p_i(t) - p_j(t))^2 = \textit{constant}$ for every $(i, j) \in E$. Thinking of t as time, and differentiating the edge length constraints we find that

$$(v_i - v_j) \cdot (p_i - p_j) = 0 \quad \text{for every } (i, j) \in E, \tag{1}$$

where v_i is the instantaneous velocity of vertex i. An assignment of velocities that satisfies (1) for a particular framework is an *infinitesimal motion* of that framework. Clearly, the existence of a finite flexing implies an infinitesimal motion, but the converse is not always true. However, for generic realizations infinitesimal motions always correspond to finite flexings [33].

The infinitesimal motions of a framework constitute a vector space. Note that a motion of the Euclidean space itself, a rotation or translation, satisfies the definition of a finite flexing. Such finite flexings are said to be *trivial*. In d-space there are d independent translations and $d(d-1)/2$ rotations. If a framework has a nontrivial infinitesimal motion, it is *infinitesimally flexible*. Otherwise it is *infinitesimally rigid*. As noted above, for generic realizations infinitesimal motions correspond to finite flexings. Since we are considering only generic realizations we will drop the prefix and refer to frameworks as either rigid or flexible.

We would like to be able to determine whether a particular framework is rigid or flexible. Conveniently, this is substantially a property of the underlying graph, as the following theorem indicates [18].

THEOREM 2.1 (Gluck). *If a graph has a single infinitesimally rigid realization, then all its generic realizations are rigid.*

This theorem is critical for a graph theoretic approach to the realization problem. The frameworks built from a graph are either all infinitesimally flexible or almost all rigid. This allows for the characterization of graphs as either rigid or flexible according to the typical behavior of a framework, without reference to a specific realization. It also allows us to be somewhat cavalier in the distinction between rigid frameworks and graphs that have rigid realizations. Henceforth such graphs will be referred to as *rigid graphs.*

How can a rigid graph be recognized? Clearly, graphs with many edges are more likely to be rigid than those with only a few. In some sense the edges are constraining the possible movements of the vertices. In d-space a set of n vertices has nd possible independent motions. However, a d-dimensional rigid body in d-space has d translations and $d(d-1)/2$ rotations. (If the body has dimension $d' < d$ then it has only $d'(2d-d'-1)/2$ rotations. This corresponds to a framework with only $d'+1$ vertices.) The total number of allowed motions is the number of total degrees of freedom, nd, minus the number of rigid body motions. For convenience, we will call this quantity $S(n,d)$, where

$$S(n,d) = \begin{cases} nd - d(d+1)/2 & \text{if } n \geq d, \\ n(n-1)/2 & \text{otherwise.} \end{cases}$$

If each edge adds an independent constraint, then $S(n,d)$ edges should be required to eliminate all nonrigid motions of the graph. This intuition is sound, as the theorems in this section will demonstrate.

Any realization of a flexible graph has a nontrivial infinitesimal motion. An infinitesimal motion is a solution for velocities in (1). The matrix of this set of equations is the *rigidity matrix.* It has m rows and nd columns. Each row corresponds to an edge while each column corresponds to a coordinate of a vertex. Each row has $2d$ nonzero elements, one for each coordinate of the vertices connected by the corresponding edge. The nonzero values are the differences in the coordinate values for the two vertices. For example, consider the graph K_3, the complete graph on three vertices, positioned in $\mathbb{R}^2$. If the realization maps the vertices to locations $(0,1)$, $(-1,0)$, and $(1,0)$, the rigidity matrix would be:

$$\begin{array}{c} \\ e_{1,2} \\ e_{1,3} \\ e_{2,3} \end{array} \begin{array}{c} \begin{array}{cccccc} v_1^x & v_1^y & v_2^x & v_2^y & v_3^x & v_3^y \end{array} \\ \begin{pmatrix} 1 & 1 & -1 & -1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & -2 & 0 & 2 & 0 \end{pmatrix} \end{array}.$$

The rank of the rigidity matrix is closely related to the rigidity behavior of the framework, as this section will elucidate.

THEOREM 2.2. *A framework $p(G)$ is rigid if and only if its rigidity matrix has rank exactly equal to $S(n,d)$.*

Proof. All infinitesimal motions must be in the null space of M since the rigidity matrix expresses all constraints on the infinitesimal velocities. By construction, $S(n,d)$ is the size of the rigidity matrix minus the number of trivial infinitesimal motions. If the null space of M contains any nontrivial infinitesimal motions, then the rank must be less than $S(n,d)$. □

So the question of whether a framework is flexible can be reduced to a question about the rank of the rigidity matrix. The framework is rigid if and only if the rank of the rigidity matrix is maximal, $S(n,d)$.

THEOREM 2.3. *Every rigid framework $p(G)$ has a rigid subframework with exactly $S(n,d)$ edges.*

Proof. The rigidity matrix has rank $S(n,d)$ and each of its rows corresponds to an edge. Simply discard redundant rows and the corresponding edges until only S remain. □

COROLLARY 2.4. *For a framework $p(G)$, if $m > S(n,d)$, then there is linear dependence among the rows of the rigidity matrix.*

Proof. The maximum rank of the rigidity matrix is $S(n,d)$. □

Dependence among rows in the rigidity matrix can be expressed in terms of a matroid [30], [15]. For our purposes it will be sufficient to say that a set of edges is *independent* if their rows in the rigidity matrix are linearly independent in a generic realization. A rigid graph has $S(n,d)$ independent edges.

THEOREM 2.5. *If a framework $p(G)$ with exactly $S(n,d)$ edges is rigid, then there is no subgraph $G' = (V', E')$ with more than $S(n', d)$ edges, where $n' = |V'|$.*

Proof. Since there are only $S(n,d)$ edges, their rows in the rigidity matrix must all be independent by Theorem 2.3. But if G' has $|E'| > S(n',d)$, then by Corollary 2.4 there must be linear dependence among these edges, which is a contradiction. □

Theorems 2.3 and 2.5 say that a rigid graph with n vertices must have a set of $S(n,d)$ *well-distributed* edges, where well-distributed means that no subgraph with n' vertices has more than $S(n',d)$ edges. This requirement is often referred to as *Laman's condition* after Laman [28], who first articulated the two-dimensional version. This condition is necessary for a graph to be rigid in any dimension. It is sufficient in one dimension where $S = n - 1$. It is straightforward to show that this is equivalent to requiring the graph to be connected. Laman was able to show that it is also sufficient in two dimensions where $S = 2n - 3$.

THEOREM 2.6 (Laman). *The edges of a graph $G = (V, E)$ are independent in two dimensions if and only if no subgraph $G' = (V', E')$ has more than $2n' - 3$ edges.*

COROLLARY 2.7. *A graph with $2n - 3$ edges is rigid in two dimensions if and only if no subgraph G' has more than $2n' - 3$ edges.*

This was the first graph theoretic characterization of rigid graphs in two-space. Several equivalent characterizations have since been discovered [36], [24], [30], [37], [12].

Unfortunately, for all its intuitive appeal Laman's condition is not sufficient in higher dimensions. A three-dimensional counterexample is depicted in Fig. 3. Although this graph has the required 18 well-distributed edges, it is still flexible. The top and bottom halves can pivot about the left- and right-most vertices.

The problem with Fig. 3 is that its edges are not independent in the sense of Theorem 2.2. The rows of the rigidity matrix are linearly dependent. Expressing this independence graph theoretically has proved to be a very difficult problem. No general characterization of rigid graphs in three dimensions is known, although the problem has been considered by many researchers, and several special cases have been solved. Cauchy proved that triangulated planar graphs (those with all $3n - 6$ edges) are generically rigid in three-space [6]. Fogelsanger recently generalized this result to include complete triangulations of any two-manifold in three-space [14]. Another class known to be rigid is complete bipartite graphs with at least five vertices in each vertex set [5], [38]. However, the characterization of general graphs remains open.

Recent work by Tay and Whiteley [37] has brought such a characterization almost within reach. However, it is difficult to see how this possible solution could lead to an efficient algorithm. Any straightforward implementation of their approach would

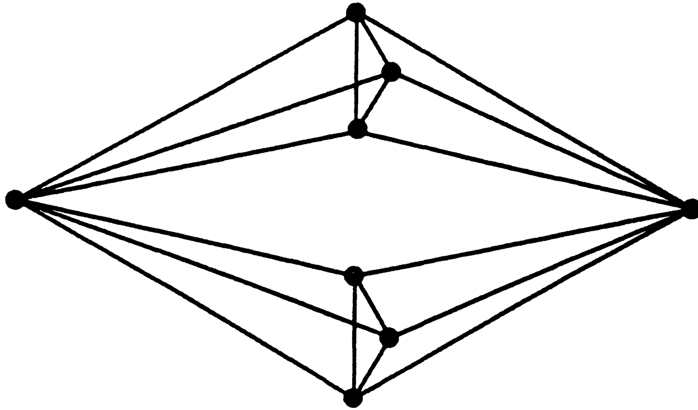


FIG. 3. *A flexible graph in three-space that satisfies Laman's condition.*

have a worst-case exponential time behavior.

2.2. Algorithms for rigidity testing. In one-space, rigidity is equivalent to connectivity. There are simple connectivity algorithms that run in time proportional to the number of edges in the graph [1].

2.2.1. Rigidity algorithms in two dimensions. In two dimensions Laman's condition characterizes rigidity, but in its original form it gives a poor algorithm. It involves counting the edges in every subgraph, of which there are an exponential number. Sugihara discovered the first polynomial time algorithm for determining the independence of a set of edges in two dimensions [36]. Imai presented an $O(n^2)$ algorithm for rigidity testing using a network flow approach [24]. This time complexity was matched by Gabow and Westermann using matroid sums [15]. In this section we will develop a new $O(n^2)$ algorithm based on bipartite matching. Besides any intrinsic interest, this new algorithm will be needed in §4 when we need to test for a stronger graph condition.

We will first need to introduce a particular bipartite graph $B(G)$ generated by our original graph $G = (V, E)$. The bipartite graph has the edges of G as one of its vertex sets, and two copies of the vertices of G for the other. Edges of $B(G)$ connect the edges of G with the two copies of their incident vertices. More formally, $B(G) = (V_1, V_2, \mathcal{E})$, where $V_1 = E$, $V_2 = \{q_1^1, q_1^2, \cdots, q_n^1, q_n^2\}$, and $\mathcal{E} = \{(e, q_i^1), (e, q_i^2), (e, q_j^1), (e, q_j^2) : e = (v_i, v_j) \in E\}$. $B(G)$ has $2n + m$ vertices and $4m$ edges, where n and m are, respectively, the number of vertices and edges in G. A simple example of the correspondence between G and $B(G)$ is presented in Fig. 4 for the graph K_3.

This bipartite graph leads to an alternate form of Laman's condition, expressed in the following theorem. As above, a set of edges is said to be *independent* if the corresponding rows in the rigidity matrix are linearly independent in a generic realization.

THEOREM 2.8. *For a graph $G = (V, E)$ the following are equivalent:*

(A) *The edges of G are independent in two dimensions;*

(B) *For each edge (a, b) in G, the graph $G_{a,b}$ formed by adding three additional edges between a and b has no subgraph G' in which $m' > 2n'$;*

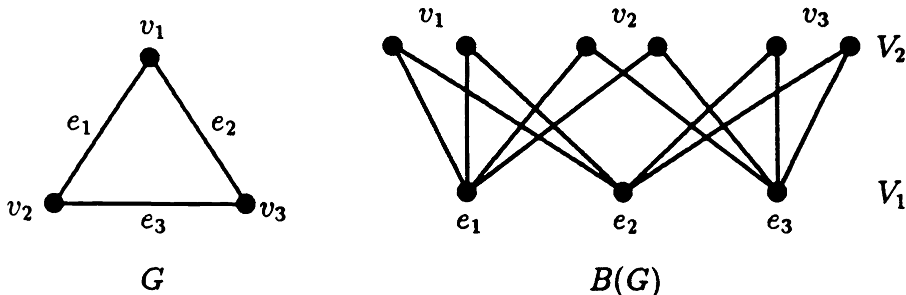


FIG. 4. *The correspondence between G and $B(G)$.*

(C) *For each edge (a, b), the bipartite graph $B(G_{a,b})$ generated by $G_{a,b}$ has no subset of V_1 that is adjacent only to a smaller subset of V_2.*

(D) *For each edge (a, b), the bipartite graph $B(G_{a,b})$ generated by $G_{a,b}$ has a complete matching from V_1 to V_2.*

Proof. The equivalence of A and B is a restatement of Laman's condition. The equivalence of B and C is a straightforward consequence of the construction of $B(G_{a,b})$. Property D is equivalent to C by Hall's theorem from matching theory. Assertions C and D were first discovered in a slightly different form by Sugihara [36]. □

Our algorithm will be based upon the characterization in Theorem 2.8(D). The basic idea is to grow a maximal set of independent edges one at a time. Denote these *basis* edges by $\hat{E}$. A new edge is added to the basis if it is discovered to be independent of the existing set. If $2n-3$ independent edges are found, then the graph is rigid. Determining whether a new edge is independent of the current basis can be done quickly using the bipartite matching characterization.

Assume we have a (possibly empty) set of independent edges $\hat{E}$. Combined with the vertices of G these form a graph $\hat{G}$, which generates a bipartite graph $B(\hat{G})$. Note that $|\hat{E}| = O(n)$ and so $B(\hat{G})$ will have $O(n)$ edges. We wish to determine if another edge, e, is independent of $\hat{E}$. Adding e to $\hat{G}$ produces $\bar{G}$ and $B(\bar{G})$. By characterization (D), e is independent of $\hat{E}$ if and only if there is a complete bipartite matching in B after any edge in $\bar{G}$ is quadrupled. Actually, only e needs to be quadrupled, as the following Lemma demonstrates.

LEMMA 2.9. *If a complete matching exists when e is quadrupled, then e is independent of $\hat{E}$.*

Proof. Assume the matching succeeds but e is not independent of $\hat{E}$. Then there must exist some edge in $\hat{E}$ whose quadrupling causes G', a subgraph of $\bar{G}$, to have $m' > 2n'-3$. Since the edges of $\hat{E}$ are independent, this bad subgraph must include e. But this bad subgraph has the same number of edges it had when e was quadrupled. Since the matching succeeded when e was quadrupled, we have a contradiction. □

Determining whether a new edge can be added to the set of independent edges is now reduced to the problem of trying to enlarge a bipartite matching. This is a standard problem in matching and it is performed by growing Hungarian trees and looking for augmenting paths. The basic idea is to look for a path from an unmatched vertex in V_1 to an unmatched vertex in V_2 that alternates between edges that are not in the current matching and edges that are. When such an augmenting path is found the matching can be enlarged by changing the unmatched edges in

the path to become matching edges, and vice versa. These paths can be found by growing Hungarian trees from the unmatched vertices in V_1. These trees grow along the unmatched edges from the starting vertex to its neighbor set in V_2. Matching edges are followed back to V_1 and unmatched edges back to V_2. If an unmatched vertex in V_2 is ever encountered, an augmenting path has been identified. Growing a Hungarian tree takes time proportional to the number of edges in the bipartite graph.

LEMMA 2.10. *If $\hat{E}$ is independent and a corresponding matching in $B(\hat{G})$ is known, then determining whether a new edge is independent requires $O(n)$ time.*

Proof. By Lemma 2.9, testing for independence of e requires just enlarging the matching in $B(\hat{G})$ to include the four copies of e. This involves growing four Hungarian trees in a bipartite graph of size $O(n)$. □

This gives a two-dimensional rigidity testing algorithm that runs in time $O(nm)$. Build a maximal set of independent edges one at a time by testing each edge for independence. Each test involves the enlargement of a bipartite matching requiring $O(n)$ time. If the matching succeeds, the edge is independent and is added to the basis. Otherwise it is discarded.

To improve this to $O(n^2)$ we need to make use of failed matchings to eliminate some edges from consideration. Define a *Laman subgraph* as a subgraph with n' vertices and $2n' - 3$ independent edges. A matching will fail precisely when the new edge lies in a subgraph that already has $2n' - 3$ independent edges. No edge can be added between vertices in this subgraph, so it is a waste of time even to try. By avoiding these unnecessary attempts, we can improve the performance of our algorithm. To accomplish this we will need some further insight into the bipartite matching.

THEOREM 2.11. *In a bipartite graph $(V_1, V_2, \mathcal{E})$, if a Hungarian tree fails to find an alternating path, then it spans a minimal subgraph that violates Hall's theorem. That is, it identifies a minimal set of k vertices in V_1 with fewer than k neighbors.*

Proof. The proof is a simple consequence of Hall's theorem. □

LEMMA 2.12. *If the new edge e is tripled instead of quadrupled, generating a graph $\underline{G}$ from $\hat{G}$, then $B(\underline{G})$ has a complete matching.*

Proof. Assume the contrary. Then there is some subgraph G' of $\bar{G}$ with $m' > 2n'$. Remove the three copies of e from this subgraph and quadruple one of the other edges. This altered subgraph still has $m' > 2n'$, but it is the graph generated by quadrupling an edge in $\hat{G}$. But since the edges of $\hat{G}$ are assumed to be independent, this is a contradiction. □

LEMMA 2.13. *If edge e fails the matching test, then the failing Hungarian tree spans a set of edges of $\hat{E}$ that form a Laman subgraph.*

Proof. By Lemma 2.12, when e is quadrupled the first three copies of it can be matched. By Theorem 2.11, when the fourth fails it spans a set of vertices of V_1 adjacent to a smaller set from V_2. Discarding the four copies of e leaves a set of k elements of V_1 adjacent to no more then $k + 3$ vertices from V_2. By the construction of the bipartite graph this is a set $\hat{E}'$ of k edges of $\hat{E}$ incident upon no more than $(k + 3)/2$ vertices. That is, $m' \geq 2n' - 3$. Since the edges of $\hat{E}$ are independent, we must have equality. □

We will need the following result to analyze the running time of our algorithm.

LEMMA 2.14. *Let $G = (V, \hat{E})$ be a graph whose edges are independent. If two Laman subgraphs of G share an edge, then their union is a Laman subgraph.*

Proof. Let the subgraphs be (V', E') and (V'', E'') with union $(\bar{V}, \bar{E})$. Let $\bar{m} = m' + m'' - l$ and $\bar{n} = n' + n'' - k$. Since the subgraphs share at least one independent

edge, $l \leq 2k - 3$. Hence,

$$\begin{aligned} \bar{m} &= 2n' + 2n'' - 6 - l \\ &\geq 2n' + 2n'' - 2k - 3 \\ &= 2(n' + n'' - k) - 3 \\ &= 2\bar{n} - 3. \end{aligned}$$

Since the edges are independent, we must have equality. □

We are now ready to present our algorithm. We will maintain the appropriate bipartite graph with a matching of all the independent edges discovered so far. We will also keep a collection of all the Laman subgraphs that have been identified, represented as linked lists of independent edges. The algorithm is outlined in Fig. 5.

```
basis ← ∅
For Each vertex v
    Mark each vertex in a Laman subgraph with v, and unmark all others
    For Each edge (u, v)
        If u is unmarked Then
            If (u, v) is independent of basis Then
                add (u, v) to basis
                create Laman subgraph consisting of (u, v)
            Else a new Laman subgraph has been identified
                Merge all Laman subgraphs that share an edge
                Mark each vertex in a Laman subgraph with v
```

FIG. 5. *An $O(n^2)$ algorithm for two-dimensional graph rigidity.*

By Lemma 2.14 we know that no edge need be in more than one subgraph. By merging whenever a new Laman subgraph is found, we guarantee that the total number of elements in all the subgraphs is kept to $O(n)$. This ensures that the marking and merging operations can be done in $O(n)$ time. As above, checking for independence of (u, v) requires $O(n)$ time. Each time an edge is checked it results in either a new basis edge or a merging of components, so this can only happen $O(n)$ times. Hence the total time for the algorithm is $O(n^2)$.

2.2.2. Rigidity algorithms in higher dimensions. For dimensions greater than two there are no graph theoretic characterizations of rigidity, so there are no good combinatoric algorithms to test for it. One approach would involve a symbolic calculation of the rank of the rigidity matrix by symbolically constructing the determinant. However, the determinant can have an exponential number of terms, so this requires an exponential amount of time. A different approach is possible that relies instead upon Theorem 2.1. Since this theorem is valid in all dimensions, the following discussion is applicable to all spaces. If the graph is rigid, then almost any realization will generate a rigid framework. Simply select a random realization for the graph. Once these vertex locations are selected it is a straightforward matter to determine the rigidity of the framework using Theorem 2.2. Just construct the rigidity matrix M and determine its rank. If the rank is $S(n, d)$, then the graph is rigid. A lower rank indicates that the framework is flexible. Unless the selection of vertex coordinates was extremely unlucky the underlying graph will be flexible as well. So even without a graph theoretic characterization an efficient practical randomized algorithm for rigidity exists.

To determine the rank of M we suggest using a QR decomposition with column pivoting, requiring $O(mn^2)$ time [19]. This is more numerically stable than Gaussian elimination, but not as costly as a singular value decomposition. A QR factorization has several advantages over an SVD in this application. Performing a QR on M^T will identify a maximal independent set of rows of M one at a time, corresponding to a maximal set of independent edges in the graph. This ability to identify independent rows will be needed in §4. Also, the rigidity matrix is quite sparse, having only $2d$ nonzeros in each row. To save time and space, sparse techniques could be used for large problems. There are sparse QR algorithms, but none for SVD [8], [16], [17].

There are also efficient parallel algorithms for finding the rank of a matrix. Ibarra, Moran, and Rosier [23] discovered an algorithm that runs in $O(\log^2 m)$ time on $O(m^4)$ processors. This means that rigidity testing is in random NC for any dimension. The class NC is the set of problems that can be solved in polylogarithmic time using a polynomial number of processors. It is a standard measure of a *good* parallel algorithm, although its applicability is more theoretical than practical.

3. Partial reflections. Even rigid graphs can have multiple realizations as was shown in Fig. 2. This discontinuous flavor of nonuniqueness has not been well studied, probably because it is not relevant to structural engineers. Buildings can only deform continuously. For the graph realization problem these discontinuous transformations must be considered. This section and the next will be concerned with multiple realizations that do not arise from flexibility. These are cases in which there are two or more noncongruent realizations that satisfy all the distance requirements, but there is no continuous flexing of the framework to transform one to another while maintaining the constraints. Whereas flexible graphs have an infinite number of potential configurations, the number of realizations of a rigid graph is finite, although possibly exponential in the size of the graph.

A two-dimensional example of the simplest type of discontinuous transformation is depicted in Fig. 2. The right half of this graph is able to fold across the line formed by the two middle vertices. When can this type of nonuniqueness occur? As in Fig. 2, there must be a few vertices about which a portion of the graph can be *reflected.* These vertices form a *mirror.* There must be no edges between the two halves of the graph separated by this mirror. For the general d-dimensional problem, the mirror vertices must lie in a $(d-1)$-dimensional subspace. We will say that a framework in d-space allows a *partial reflection* if a separating set of vertices lies in a $(d-1)$-dimensional subspace.

The realizations in which more than d vertices lie in a $(d-1)$-dimensional subspace are not generic. So for almost all frameworks, partial reflections only occur when there is a subset of d or fewer vertices whose removal separates the graph into two or more unconnected pieces, that is, when G is not vertex $(d+1)$-connected. This gives us the following well-known result.

THEOREM 3.1. *A rigid graph positioned generically in dimension d will have a partial reflection if and only if it is not vertex $(d+1)$-connected.*

The connectivity of a graph is an important property, and it has been well studied. There are well-known $O(m)$ time algorithms for vertex two-connectivity, also known as *biconnectivity* [1]. Avoiding partial reflections in two dimensions requires a vertex three-connected (or *triconnected*) graph. Hopcroft and Tarjan [22] were the first to discover an $O(m)$ time algorithm to find triconnected components. Miller and Ramachandran [31] have recently proposed a parallel algorithm to identify triconnected components in $O(\log^2 n)$ time with $O(m)$ processors, placing triconnectivity in NC.

Four-connected components are more difficult, but Kanevsky and Ramachandran [25] have recently found an $O(n^2)$ time algorithm. They also discovered a parallel implementation of their algorithm that runs in $O(\log n)$ time using $O(n^2)$ processors. So the problem of partial reflections is in NC in both two and three dimensions.

For k greater than 4, the question of k-connectivity for a general k is less well understood. Consequently the partial reflection problem is more difficult in spaces of dimension greater than three. There are randomized algorithms for general k-connectivity that run in time proportional to $n^{5/2}$ [4], [29]. Recently, Cheriyan and Thurimella have described an algorithm with a time complexity of $O(k^3n^2)$, which reduces to $O(n^2)$ for a fixed k [7]. There are also NC algorithms that run in time $O(k^2 \log n)$ [26].

4. Redundant rigidity. Rigidity and $(d+1)$-connectivity are necessary but not sufficient to ensure that a graph has a unique realization. A two-dimensional example of a rigid, triconnected graph with two satisfying realizations is given in Fig. 6.

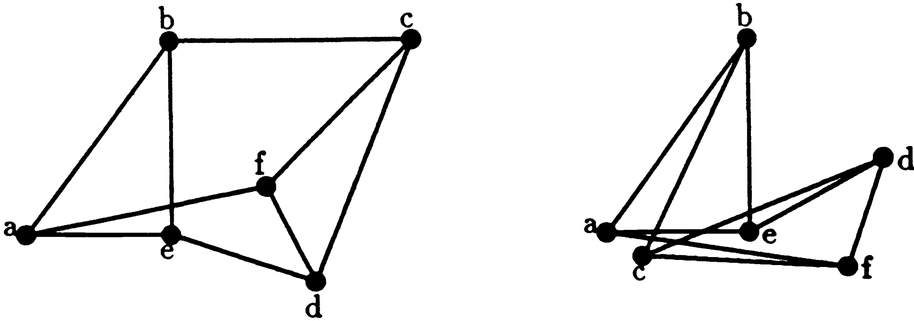


FIG. 6. *Two realizations of a rigid triconnected graph in the plane.*

To understand this nonuniqueness, consider Fig. 7. Edge (a, f) has been removed from the original graph. This resultant graph is now composed of a quadrilateral *bcde* with two attached triangles *abe* and *cdf*. The quadrilateral is not rigid, so this new graph can flex. The flexing will lift vertex d up until it crosses the line between c and e as depicted in the center picture of Fig. 7. Eventually vertex c can swing all the way around to the right. As the graph moves, the distance between vertices a and f varies. When vertex c swings far enough around, this distance becomes the same as it was originally, as shown in the rightmost picture of Fig. 7. Now the missing edge can be replaced to yield a new satisfying realization.

The fundamental problem with the graph in Fig. 6 is that the removal of a single edge makes it flexible. We will define an edge of a framework to be *infinitesimally redundant* if the framework remaining after its removal is infinitesimally rigid. A framework is *infinitesimally redundantly rigid* if all its edges are infinitesimally redundant. Redundant bracing is a familiar concept to engineers who wish to build frameworks with additional strength and failure tolerance properties, but this precise formulation and its significance with regards to the unique realization problem are entirely new.

Infinitesimal redundant rigidity is clearly a more restrictive property than infinitesimal rigidity, but the two properties have many similarities. For generic realizations infinitesimal motions always correspond to finite flexings. So as with rigidity,

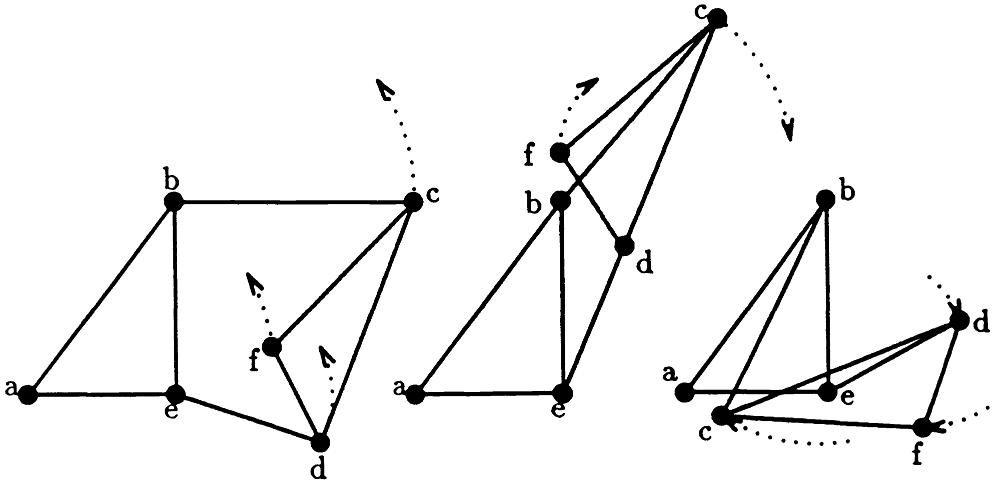


FIG. 7. *Intermediate stages in the construction of Fig.* 6.

since we are only interested in generic realizations we can drop the prefix and refer to frameworks as redundantly rigid. The following theorem is a trivial consequence of Theorem 2.1.

THEOREM 4.1. *If a graph has a single infinitesimally redundantly rigid realization, then all its generic realizations are redundantly rigid.*

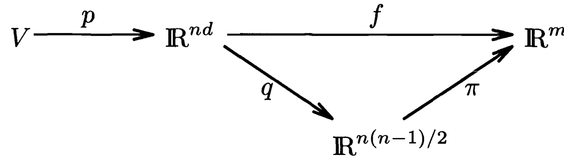
As with Theorem 2.1 for rigid graphs, Theorem 4.1 says that either none of a graph's realizations are redundantly rigid, or almost all of them are. *Almost all* again means that the set of counterexamples has measure zero. This blurs the distinction between a redundantly rigid framework and its underlying graph. Graphs with redundantly rigid realizations will be referred to as *redundantly rigid graphs.*

In Fig. 6 the lack of redundant rigidity led to multiple realizations. This is usually true, and the proof will be the main result of this section. Intuitively, a flexible framework can move around, but it must always end up back where it started. That is, the path it traces in nd-space will be a loop. If the removal of an edge allows the graph to flex, then the distance corresponding to that edge must be a multivalued function as the flexing completes its loop. However, there are some graphs for which this argument fails. Consider the triangle graph K_3. It has only one realization, but if an edge is removed it becomes flexible. To understand which graphs need to be redundantly rigid to have unique realizations, we will need to carefully investigate the space of satisfying realizations for flexible graphs. This will require an incursion into differential topology, and is the subject of the next section.

5. The necessity of redundant rigidity. The proof that flexible graphs typically move in closed loops will rely upon some special properties of the graph realization problem. Given a framework $p(G)$ there is a *pairwise distance function* $q : \mathbb{R}^{nd} \rightarrow \mathbb{R}^{n(n-1)/2}$ that maps vertex locations to squares of all the pairwise vertex distances. That is,

$$q(p(v_1), \cdots, p(v_n)) = (\cdots, |p(v_i) - p(v_j)|^2, \cdots).$$

For the molecule problem, we are only interested in a specific set of pairwise distances, namely, those corresponding to the edges of G. These can be obtained from $R^{n(n-1)/2}$ by a simple projection π. We will define an *edge function* f to be the composition of these two operations, $f = \pi \circ q$. These functions are described by the following commutative diagram.



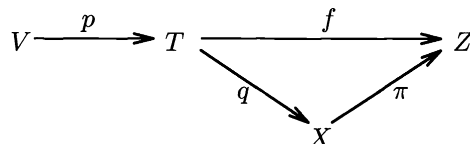
The functions f and q have many nice properties. We will say a function is *smooth* at a point x if it has continuous partial derivatives of all orders at x. The functions f and q are everywhere smooth. Also, the Jacobian of f is twice the rigidity matrix introduced in §2.1.

The realization problem is really that of finding the inverse of the edge function. Of course, this inverse is multivalued because edge lengths are invariant under translations, rotations, and reflections of the entire space. Two realizations will be considered *equivalent* if all pairwise distances between vertices are the same under the two realizations. That is, two realizations are equivalent if they map to the same point under q. We will be interested only in the inverse of f modulo equivalences. More formally, define the *realization set* of $p(G)$ to be $\pi^{-1}f(p(G))$, the set of nonequivalent, satisfying realizations for the graph realization problem generated by $p(G)$. For $p(G)$ to be a unique solution to the realization problem it is necessary and sufficient that this realization set consist of a single point. Our goal in this section is to investigate the structure of the realization set. Our first result is the following theorem.

THEOREM 5.1. *If a graph G is connected, then the realization set of $p(G)$ is compact.*

Proof. The realization set is a subset of $\mathbb{R}^{n(n-1)/2}$. It is bounded since the graph is connected, and it is trivially closed. □

Although every point in $\mathbb{R}^{nd}$ corresponds to a realization, the image of $\mathbb{R}^{nd}$ under q does not cover $\mathbb{R}^{n(n-1)/2}$. Define this image to be a space $W \subset \mathbb{R}^{n(n-1)/2}$. The space W has a natural topology and measure inherited from the larger Euclidean space. For technical reasons we will restrict our consideration of realizations to those in which not all the vertices lie in a hyperplane. Call this subset of realizations T. The space T is a dense, open subset of $\mathbb{R}^{nd}$. Define X to be the subset of points in W that are images of points in T under q. If the graph has d or fewer vertices, then X is empty. Otherwise, X is a dense, open subset of W, with a nice structure, as we will see shortly. Define Z to be the image of X under π. This gives us the following structure.



We will need the following notation from differential topology. Say the largest rank the Jacobian of a function $g : A \to B$ attains in its domain is k. A point $x \in A$ is called a *regular point* if the Jacobian of g at x has rank k. A point $y \in B$ is a *regular value* if every point in the preimage of y under g is a regular point. If a point or value is not regular, it is *singular*. Note that for the edge function singular points are not generic. A *j-dimensional manifold* is a subset of some large Euclidean space that is everywhere locally diffeomorphic to $\mathbb{R}^j$.

Consider the following procedure for identifying equivalent realizations, which is defined for any realization in T. Select a set of $d+1$ vertices from $p(G)$ whose affine span is all of $\mathbb{R}^d$. Translate the realization so that the first of these vertices is at the origin. Next rotate about the origin to move the second of these vertices onto the positive x_1 axis. Now rotate, keeping the first two vertices fixed, to move the third to the (x_1, x_2) plane so that the x_2 coordinate is positive. Continuing this process in the obvious way gives a smooth mapping that makes $d(d+1)/2$ of the vertex coordinates zero. Finally, if the $d+1$st vertex has its $d+1$st coordinate less than zero, reflect the vertices through the hyperplane defined by the $x_1, \cdots, x_{d-1}$ axes.

This procedure maps all equivalent realizations to a single one. This single realization can be described by its remaining variable coordinates, of which there are $nd - d(d+1)/2$. Since each of these remaining coordinates can vary continuously, the realization can be considered to be a point in $\mathbb{R}^{nd-d(d+1)/2}$. This defines a coordinate chart for X. Note that the sequence of operations performed on the original realization is smooth and invertible. If a different set of $d+1$ initial vertices was selected, a different coordinate chart would have been generated. Since these coordinate transformations are smooth and invertible, on regions of intersection the two charts are diffeomorphic. The union of all such charts gives a differentiable structure to our space X. This construction provides a diffeomorphism between each open set of a collection that covers X and $\mathbb{R}^{nd-d(d+1)/2}$, giving us the following theorem.

THEOREM 5.2. *If the graph has at least $d+1$ vertices, then X is a smooth manifold of dimension $nd - d(d+1)/2$.*

The dimension of this manifold is a quantity that will come up frequently, so it will be convenient to reintroduce the following notation: $S(n,d) = nd - d(d+1)/2$. This function was first defined in §2.1 as the maximal rank of the rigidity matrix of a graph with n vertices positioned in d-space.

The procedure described above gives us an alternate way in which to view the space X. The sequence of translations, rotations, and reflections constitute a function $\bar{q}$ that maps an entire set of equivalent realizations to a single one. The remaining variable coordinates uniquely define a point in X. Considering these to be the independent variables, the mapping from X to Z becomes more complicated than a simple projection. We will define this function to be $\bar{f}$, giving us the following commutative diagram.

$$
\begin{array}{ccccc}
V & \xrightarrow{\;p\;} & T & \xrightarrow{\qquad f \qquad} & Z \\
 & & & \searrow^{\bar{q}} \quad \nearrow_{\bar{f}} & \\
 & & & X &
\end{array}
$$

This function $\bar{f}$ is closely related to the edge function f. In fact, $\bar{f}$ is everywhere smooth in X, and the rank of the Jacobian of $f(x)$ is the same as that of $\bar{f}(\bar{q}(x))$. So the singular values of f are the same as the singular values of $\bar{f}(\bar{q})$. If we designate the

number of independent edges of G by k, then the rank of these Jacobians is almost always k. The following is a special case of a well-known theorem due to Sard [34].

THEOREM 5.3 (Sard). *The set of singular values of f has k-measure zero.*

LEMMA 5.4. *If Z' is a subset of Z with k-measure zero, then for almost all realizations p, $f(p) \notin Z'$.*

Proof. The singular points of f constitute an algebraic variety in $\mathbb{R}^{nd}$ with dimension less than nd. Hence, the regular points of f can be covered by a countable number of open neighborhoods in such a way that the rank of the Jacobian of f is maximal within each neighborhood. Consider one of these neighborhoods $\mathcal{R}$, and let its image under f be $\mathcal{Z}$. By the implicit function theorem from analysis, there is a submersion from $\mathcal{R}$ to $\mathcal{Z}$. That is, on this neighborhood f is diffeomorphic to a projection from $\mathbb{R}^{nd}$ to $\mathbb{R}^k$. Since Z' has k-measure zero its inverse image under this submersion must have (nd)-measure zero in $\mathcal{R}$. The countable union of these sets of (nd)-measure zero yields a preimage for Z' with (nd)-measure zero. □

These last two results imply the following theorem.

THEOREM 5.5. *For almost every realization p, $f(p)$ is a regular value.*

All this has been leading up to the following crucial result.

THEOREM 5.6. *For almost every realization p, the realization set of $p(G)$ restricted to X is a manifold.*

Proof. Almost all realizations map to regular values of f and hence of $\bar{f}(\bar{q})$. The preimage of a regular value is a submanifold of X by the implicit function theorem from differential topology [20]. □

If the graph is flexible, then this manifold describes the allowed flexings. At any point in the manifold, the tangent space is exactly the null space of the Jacobian of $\bar{f}$. To show that flexings typically move in closed loops (actually, one-manifolds diffeomorphic to the circle), we will need the flexings to remain entirely in our manifold X. This can be ensured if the graph has *enough* independent edges. Enough means more than can be independent in a lower-dimensional space, as the following theorem demonstrates.

THEOREM 5.7. *If G has more than $S(n, d-1)$ independent edges, then for almost all realizations $p(G)$, the realization set of $p(G)$ stays within X.*

Proof. Assume the theorem is false. By the definition of X this means that the realization set must include a point at which all the vertices lie in a $(d-1)$-dimensional hyperplane. When this happens the edges can only constrain infinitesimal motions within the hyperplane. The rows of the rigidity matrix describe these infinitesimal constraints, so when the vertices lie in a hyperplane the rank of the rigidity matrix can be no larger than $S(n, d-1)$. Since there are more than $S(n, d-1)$ independent edges, this implies that $f(p(G))$ is a singular value, but by Theorem 5.5 this cannot be the case for almost all realizations. □

We can finally prove that flexings typically move in closed loops.

THEOREM 5.8. *If a graph G is connected, flexible, and has more than $d+1$ vertices, then for almost all realizations $p(G)$ the realization set of $p(G)$ contains a submanifold that is diffeomorphic to the circle.*

Proof. Generate a new graph G' from G by arbitrarily adding additional edges until G' has $S(n, d) - 1$ independent edges. The realization set of $p(G')$ must be a subset of the realization set of $p(G)$. Since $n > d+1$, it is easy to show that the number of independent edges is now greater than $S(n, d-1)$. By Theorems 5.6 and 5.1 we know that for almost all realizations the realization set of $p(G')$ is a compact manifold of dimension one. It is a well-known result from differential topology that

such manifolds are diffeomorphic to the circle. □

This finally leads us to the main result of this section.

THEOREM 5.9. *If G is not redundantly rigid and G has more than $d+1$ vertices, then almost all realizations of G are not unique.*

Proof. Assume the only interesting case, that G is rigid. Then the graph G must have $S(n,d)$ independent edges, and there is some edge e_{ij} of G whose removal generates a flexible graph G'. By Theorem 5.8, for almost all realizations p the realization set of $p(G')$ contains a submanifold diffeomorphic to the circle. The distance between vertices i and j will be a multivalued function for almost every point on this circle. The only distances that might not be multivalued are the extremal ones. When a flexing reaches a realization that induces an extremal value between i and j, the derivative of $d_{i,j}^2$ is zero in the direction of the flex. In this case the realization is not generic [32]. So almost all realizations do not induce extremal edge lengths. □

Theorem 5.9 means that the example in Fig. 6 was not a fluke. Redundant rigidity is a necessary condition for unique realizability.

5.1. Algorithms for redundant rigidity. How difficult is it to test for redundant rigidity? A simplistic approach would use the algorithm for rigidity repeatedly, removing one edge at a time. This approach parallelizes easily by simply running the m different problems on independent sets of processors. Since rigidity testing was shown to be in deterministic or random NC for all dimensions, redundant rigidity is as well.

In one dimension redundant rigidity is equivalent to edge two-connectivity. This property can be determined by looking for cut points of the graph, requiring $O(m)$ time [1].

For the two-dimensional case a simple modification of the rigidity testing algorithm described in §2.2 can be employed. The rigidity algorithm grows a basis set of independent edges one at a time by checking them against the existing independent set. If a new edge is found to be independent of the existing set, then it is added. Independence is determined by the success of a particular bipartite matching. If the matching fails, then there must be some dependence among the edges. Identifying and utilizing these dependencies will lead to an efficient redundant rigidity algorithm.

As in §2.2, we will denote by $B(G)$ the bipartite graph constructed from $G = (V,E)$. The current set of independent, *basis* edges is $\hat{E}$, generating a subgraph $\hat{G} = (V,\hat{E})$. When a new edge e is to be tested for independence, four copies of it are added to $\hat{G}$, generating $\bar{G}$ with its corresponding bipartite graph $B(\bar{G})$. As we saw in §2.2, if a complete bipartite matching exists in $B(\bar{G})$, then e is independent of $\hat{E}$. For our current purposes we are interested in dependent edges and how they contribute to redundant rigidity. Dependent edges fail to have complete matchings in $B(\bar{G})$. However, if we triple e instead of quadrupling it, generating $\underline{G}$ and $B(\underline{G})$, then Lemma 2.12 guarantees that $B(\underline{G})$ always has a complete matching. So only a single vertex in $B(\bar{G})$ can go unmatched. This is important because of the following general property of bipartite matching.

THEOREM 5.10. *Let $B = (V_1, V_2, \mathcal{E})$ be a bipartite graph with a matching from V_1 to V_2 involving all but one vertex from V_1, denoted by v. Also let $\mathcal{V}_1$ be the subset of V_1 that is in the Hungarian tree built from v. Then if any vertex from $\mathcal{V}_1$ is deleted from B, the resulting graph will have a complete matching.*

Proof. The removal of a vertex w from $\mathcal{V}_1$ creates an unmatched vertex in V_2 that is reachable from v along an alternating path. □

Theorem 5.10 identifies which vertices of a bipartite graph can be removed to

result in a perfect matching. For our purposes, these are vertices in $B(\bar{G})$, which correspond to edges of $\bar{G}$. If any of these edges of $\bar{G}$ is removed, then the new edge e will be independent of the remaining basis edges. That is, e can replace any of these edges identified by the Hungarian tree, leaving the number of basis edges unchanged. More formally, we have the following theorem.

THEOREM 5.11. *In the rigidity algorithm, assume a new edge e is found to be not independent of the current set of k independent edges. Let* $\mathcal{V}_1$ *be the subset of vertices of* V_1 *that are in the Hungarian tree of the failed matching. Then if e replaces any of the edges in* $\mathcal{V}_1$ *the resulting set of k edges is still independent.*

Theorem 5.11 gives an efficient algorithm for redundant rigidity testing. An edge is not independent of the current basis set if the bipartite matching fails. When this happens the Hungarian tree identifies precisely which edges are dependent. All these edges are redundant because any of them could be replaced by the new edge. In the $O(n^2)$ algorithm from §2.2 a Laman subgraph is identified by this Hungarian tree. Hence, any edge in the Laman subgraph is redundant. When the algorithm is finished, if there is a basis edge that has not been merged into a larger Laman subgraph, then it is not redundant and the graph is not redundantly rigid. Note that if the full graph is not redundantly rigid, then the Laman subgraphs identified by this procedure are redundantly rigid components. This takes essentially no more effort than testing for rigidity, so two-dimensional redundant rigidity can be decided in $O(n^2)$ time.

In dimensions greater than two there is no graph theoretic characterization of redundant rigidity. As in §2.2 an algorithm will have to randomly position the vertices and then examine the rigidity matrix. Like the two-dimensional case, the basic idea will be to build a set of independent edges one at a time, and then determine which of them are redundant. Every time a new edge fails to be independent it supplies information about the redundancy of some of the independent edges. If a full set of redundant, independent edges are found, then the graph is redundantly rigid.

Begin by positioning the vertices randomly and constructing the rigidity matrix M. The rigidity of the framework can be determined by performing a QR factorization on M^T to find its rank. This procedure will form an independent set of edges one at a time. A new column is added if it is linearly independent of the current set of k columns; otherwise it is discarded. A discarded column, corresponding to an edge e, can be expressed as a linear combination of some set of the independent columns. The discarded column could replace any of the columns in the linear combination that forms it, without altering the span of the independent set.

How difficult is it to determine which of the current columns contribute to the linear combination? Assume the algorithm has identified k independent columns of M^T. Place these columns together to form an $nd \times k$ matrix A_k. The QR factorization has been proceeding on these columns as they are identified, so there is a $k \times k$ orthogonal matrix Q_k and an $nd \times k$ upper triangular matrix R_k satisfying $Q_k R_k = A_k$. If a new column b of M^T is linearly dependent upon the columns of A_k, then there must be a vector c satisfying $A_k c = Q_k R_k c = b$, or alternately, $R_k c = Q_k^T b$. In the course of the QR factorization the column b has been overwritten with $Q_k^T b$, so it is easy to solve the upper triangular system for c. The nonzero elements of c identify which columns of A_k contribute to the linear combination composing b, that is, which columns are redundant.

How much work does this take? There are $O(m)$ triangular systems to solve, each of which requires $O(k^2)$ operations, where k is always $O(n)$. So the total additional time is of the same order as the QR factorization itself, $O(mn^2)$. As in the two-

dimensional case, the redundant rigidity of a graph can be determined by modifying the rigidity algorithm without incurring substantial increased cost.

As was noted in §2.2, the rigidity matrix consists mostly of zeros. For large problems this property should be exploited by using sparse matrix techniques. The only real modification to the rigidity algorithm required to verify redundant rigidity is a sequence of triangular solves. These can be done sparsely, so the entire algorithm can be implemented in a sparse setting. An algorithm very similar to this has been described by Coleman and Pothen [9].

6. Conclusion. Three necessary conditions for almost all realizations of a graph to be unique in d dimensions have been derived. They are, in order of appearance, rigidity, $(d+1)$-connectivity, and redundant rigidity. The first condition is a trivial consequence of the third so there are really only two independent criteria. However, flexibility leads to a very different kind of nonuniqueness than lack of redundant rigidity, so it is useful to think of them independently. Efficient algorithms for testing each of these three conditions have been presented that deal solely with the underlying graph, ignoring the edge lengths. The price for this convenience is that there are combinations of edge lengths for which these conditions are not necessary. But these counterexamples are very rare. For almost all realizations, a graph that violates one of these conditions will have multiple satisfying realizations.

Establishing necessary conditions for a graph to have a unique realization makes it possible to prune the initial graph before attempting the difficult task of finding coordinates for the vertices. If the entire graph does not have a unique realization, then it would be impossible to assign coordinates unambiguously. Instead, portions of the graph that do satisfy the necessary criteria can be identified and positioned. Not only does this alleviate the confusion of a poorly posed problem, but since the cost of finding the realization can grow exponentially with the size of the graph, it should be possible to save time by positioning a sequence of smaller subgraphs instead of the original full one.

Following this idea to its logical conclusion, even if the original graph has a unique realization it might be possible to position subgraphs first and then piece them together. Since the running time grows rapidly with problem size, this could lead to a substantial reduction of computational effort. In fact, an approach to the molecule problem using precisely this approach has recently been proposed [21]. For this approach to be infallible we would need to develop sufficiency conditions for a graph to have a unique realization. Unfortunately, the necessary conditions developed in this paper are not sufficient. Connelly has identified a class of bipartite graphs that satisfy the conditions presented here, while still allowing multiple realizations in high-dimensional spaces [10]. There are no graphs in this class in one or two dimensions, and $K_{5,5}$ is the only example in three-space. A complete characterization of uniquely realizable graphs remains an open problem. In fact, it is also unknown whether uniqueness itself is a generic property. That is, if a single generic realization of a graph is unique, are almost all realizations unique?

Acknowledgments. The ideas in this paper have been developed and refined in innumerable discussions with Tom Coleman and Bob Connelly.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, MA, 1974.

[2] L. ASIMOW AND B. ROTH, *The rigidity of graphs*, Trans. Amer. Math. Soc., 245 (1978), pp. 279–289.
[3] ———, *The rigidity of graphs,* II, J. Math. Anal. Appl., 68 (1979), pp. 171–190.
[4] M. BECKER, W. DEGENHARDT, J. DOENHARDT, S. HERTEL, G. KANINKE, W. KEBER, K. MEHLHORN, S. NÄHER, H. ROHNERT, AND T. WINTER, *A probabilistic algorithm for vertex connectivity of graphs*, Inform. Process. Lett., 15 (1982), pp. 135–136.
[5] E. D. BOLKER AND B. ROTH, *When is a bipartite graph a rigid framework?*, Pacific J. Math., 90 (1980), pp. 27–44.
[6] A. L. CAUCHY, *Deuxième memoire sur les polygones et les polyèdres*, J. l'Ecole Polytechnique, 19 (1813), pp. 87–98.
[7] J. CHERIYAN AND R. THURIMELLA, *On determining vertex connectivity*, Tech. Report UMIACS-TR-90-79, CS-TR-2485, Department of Computer Science, University of Maryland at College Park, MD, 1990.
[8] T. F. COLEMAN, A. EDENBRANDT, AND J. R. GILBERT, *Predicting fill for sparse orthogonal factorization*, J. Assoc. Comput. Mach., 33 (1986), pp. 517–532.
[9] T. F. COLEMAN AND A. POTHEN, *The null space problem* II. *Algorithms*, SIAM J. Algebraic Discrete Meth., 8 (1987), pp. 544–563.
[10] R. CONNELLY, *On generic global rigidity*, in Applied Geometry and Discrete Mathematics, the Victor Klee Festschrift, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Volume 4, P. Gritzmann and B. Sturmfels, eds., AMS and ACM, 1991, pp. 147–155.
[11] H. CRAPO, *Structural rigidity*, Topologie Structurale, 1 (1979), pp. 26–45.
[12] ———, *On the generic rigidity of plane frameworks*, Tech. Report, Bat 10, Institut National de Recherche en Informatique et en Automatique, Cedex, France, 1988.
[13] G. M. CRIPPEN AND T. F. HAVEL, *Distance Geometry and Molecular Conformation*, Research Studies Press Ltd., Taunton, Somerset, England, 1988.
[14] A. FOGELSANGER, *The generic rigidity of minimal cycles*, Ph.D. thesis, Department of Mathematics, Cornell University, Ithaca, NY, May 1988.
[15] H. N. GABOW AND H. H. WESTERMANN, *Forests, frames and games: Algorithms for matroid sums and applications*, in Proc. 20th Annual Symposium on the Theory of Computing, Chicago, IL, 1988, pp. 407–421.
[16] A. GEORGE AND M. T. HEATH, *Solution of sparse linear least squares problems using Givens rotations*, Linear Algebra Appl., 34 (1980), pp. 69–83.
[17] J. GILBERT, *Predicting structure in sparse matrix computations*, Tech. Report TR 86-750, Department of Computer Science, Cornell University, Ithaca, NY, 1986.
[18] H. GLUCK, *Almost all simply connected closed surfaces are rigid*, in Geometric Topology, Lecture Notes in Mathematics No. 438, Springer-Verlag, Berlin, 1975, pp. 225–239.
[19] G. H. GOLUB AND C. F. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 1983.
[20] V. GUILLEMIN AND A. POLLACK, *Differential Topology*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1974.
[21] B. HENDRICKSON, *The molecule problem: Determining conformation from pairwise distances*, Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1990. Tech. Report 90-1159.
[22] J. E. HOPCROFT AND R. E. TARJAN, *Dividing a graph into triconnected components*, SIAM J. Comput., 2 (1973), pp. 135–158.
[23] O. H. IBARRA, S. MORAN, AND L. E. ROSEN, *A note on the parallel complexity of computing the rank of order n matrices*, Inform. Process. Lett., 11 (1980), p. 162.
[24] H. IMAI, *On combinatorial structures of line drawings of polyhedra*, Discr. Appl. Math., 10 (1985), pp. 79–92.
[25] A. KANEVSKY AND V. RAMACHANDRAN, *Improved algorithms for graph four-connectivity*, in Proc. 28th IEEE Annual Symposium on Foundations of Computer Science, Los Angeles, October 1987, pp. 252–259.
[26] S. KHULLER AND B. SCHIEBER, *Efficient parallel algorithms for testing k-connectivity and finding disjoint s-t paths in graphs*, Tech. Report TR 90-1135, Department of Computer Science, Cornell University, Ithaca, NY, June 1990.
[27] K. KILLIAN AND P. MEISSL, *Einige grundaufgaben der räumlichen trilateration und ihre gefährlichen örter*, Deutsche Geodätische Komm. Bayer. Akad. Wiss., A61 (1969), pp. 65–72.
[28] G. LAMAN, *On graphs and rigidity of plane skeletal structures*, J. Engrg. Math., 4 (1970), pp. 331–340.
[29] N. LINIAL, L. LOVÁSZ, AND A. WIGDERSON, *A physical interpretation of graph connectivity,*

and its algorithmic applications, in Proc. 27th IEEE Annual Symposium on Foundations of Computer Science, Toronto, October 1986.

[30] L. Lovasz and Y. Yemini, *On generic rigidity in the plane*, SIAM J. Algebraic Discrete Meth., 3 (1982), pp. 91–98.

[31] G. L. Miller and V. Ramachandran, *A new graph triconnectivity algorithm and its parallelization*, in Proc. 19th ACM Annual Symposium on Theory of Computing, New York, 1987, pp. 335–344.

[32] B. Roth, *Questions on the rigidity of structures*, Topologie Structurale, 4 (1980), pp. 67–71.

[33] ———, *Rigid and flexible frameworks*, Amer. Math. Monthly, 88 (1981), pp. 6–21.

[34] A. Sard, *The measure of the critical values of differentiable maps*, Bull. Amer. Math. Soc., 48 (1942), pp. 883–890.

[35] J. B. Saxe, *Embeddability of weighted graphs in k-space is strongly NP-hard*, Tech. Report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1979.

[36] K. Sugihara, *On redundant bracing in plane skeletal structures*, Bull. Electrotech. Lab., 44 (1980), pp. 376–386.

[37] T.-S. Tay and W. Whiteley, *Generating isostatic frameworks*, Topologie Structurale, 11 (1985), pp. 21–69.

[38] W. Whiteley, *Infinitesimal motions of a bipartite framework*, Pacific J. Math., 110 (1984), pp. 233–255.

[39] W. Wunderlich, *Untersuchungen zu einem trilaterations problem mit komplaneren standpunkten*, Sitz. Osten. Akad. Wiss., 186 (1977), pp. 263–280.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 85–110, February 1992

© 1992 Society for Industrial and Applied Mathematics
009

SIMPLIFICATION OF NESTED RADICALS*

SUSAN LANDAU†

Abstract. Radical simplification is an important part of symbolic computation systems. Until now no algorithms were known for the general denesting problem. If the base field contains all roots of unity, then necessary and sufficient conditions for a denesting are given, and the algorithm computes a denesting of α when it exists. If the base field does not contain all roots of unity, then it is shown how to compute a denesting that is within one of optimal over the base field adjoining a single root of unity. Throughout this paper, a primitive lth root of unity is respresented by its symbol ζ_l, rather than as a nested radical. The algorithms require computing the splitting field of the minimal polynomial of α over k, and have exponential running time.

Key words. denesting, Galois theory, nested radicals, radical simplification

AMS(MOS) subject classification. 11Y16

1. Introduction. In his magical way, Ramanujan observed a number of striking relationships between certain nested radicals:

$$\sqrt[3]{\sqrt[3]{2}-1} = \sqrt[3]{1/9} - \sqrt[3]{2/9} + \sqrt[3]{4/9},$$
$$\sqrt{\sqrt[3]{5}-\sqrt[3]{4}} = 1/3(\sqrt[3]{2} + \sqrt[3]{20} - \sqrt[3]{25}),$$
$$\sqrt[6]{7\sqrt[3]{20}-19} = \sqrt[3]{5/3} - \sqrt[3]{2/3}.$$

These remained innocent curiosities for decades. Then symbolic computation came of age, and such manipulations assumed greater importance. Consider the equation:

$$\sqrt{5+2\sqrt{6}} = \sqrt{2}+\sqrt{3}.$$

The field

$$Q(\sqrt{5+2\sqrt{6}})$$

has

$$\{1, \sqrt{5+2\sqrt{6}}, 5+2\sqrt{6}, (\sqrt{5+2\sqrt{6}})^3\}$$

as a basis over Q. But $\{1, \sqrt{2}, \sqrt{3}, \sqrt{6}\}$ is also a basis for

$$Q(\sqrt{5+2\sqrt{6}})$$

over Q. In many cases, the first basis is preferable—it is of the form $\{1, \alpha, \alpha^2, \alpha^3\}$—but people often find the second basis easier to understand. The basis $\{1, \sqrt{2}, \sqrt{3}, \sqrt{6}\}$ is simple to manipulate. An important issue then, is the "denesting" of radicals—a term which will be precisely defined in the next section. It is also independently

* Received by the editors April 13, 1989; accepted for publication (in revised form) March 5, 1991. This research was supported by National Science Foundation grants DMS-8807202 and CCR-8802835. Part of this work was done while the author was visiting the Yale University Mathematics Department, New Haven, Connecticut 06520.

† Computer Science Department, University of Massachusetts, Amherst, Massachusetts 01003.

interesting: under what circumstances can a radical be expressed in terms of radicals with a lower depth of nesting?

In 1985, Borodin, Fagin, Hopcroft, and Tompa [3] gave an efficient algorithm for decreasing the nesting depth of a class of expressions involving square roots. Also in 1985, Zippel [18] gave some conditions under which a radical could denest. The general case remained open. It was unknown how to determine whether a radical could be denested.

We show that if the base field contains all roots of unity, then a radical can be denested if and only if there is a denesting which occurs within its splitting field. That is, a radical α can be denested over a field k containing all roots of unity if and only if there is a denesting which occurs with each term of the denesting lying within the splitting field of the minimal polynomial of α over k. If the base field does not contain all roots of unity, we show that a denesting within one of optimal can be achieved over $k(\zeta_l)$, ζ_l a primitive lth root of unity, where l is the lowest common multiple (lcm) of the exponents of the derived series of the Galois group of the splitting field of $k(\alpha)$ over k. We also show how to achieve an optimal denesting by adjoining a root of unity that is dependent on the presentation of α. We represent a primitive lth root of unity by the symbol ζ_l rather than as a nested radical itself. This presents certain problems, which we discuss in §3.

Following the measure of the size of univariate polynomials in the factoring problem, we define the size of the denesting problem to be the minimal polynomial for α over k. (In the Appendix we show how to go from the presentation as a nested radical to its minimal polynomial.) Next we show that given a radical α over a field k of characteristic 0, we can denest α in the time it takes to compute the splitting field of the minimal polynomial of α over k. In the worst case this takes exponential time.

Our paper is organized as follows: §2: Background; §3: Ambiguity; §4: Algebraic Structure; §5: The Algorithm and Running Time Analysis; §6: Conclusions; and the Appendix.

2. Background. We begin with a brief review of some algebraic concepts. The reader who is unfamiliar with this material is advised to consult [8] or [14] for algebraic number theory and Galois theory, and [13] for group theory.

Let k be a field. Throughout this paper we assume that k is of characteristic 0. An element α *is algebraic over* k if and only if α satisfies a polynomial with coefficients in k. The *degree* of α is the degree of the minimal irreducible polynomial of α over k. If L has finite dimension $[L : k]$ over a subfield k, then $L = k(\theta)$ for some θ in L, where the degree of θ over k equals $[L : k]$.

An extension field L is *algebraic over a field* k if and only if every element of L is algebraic over k. It is well known that every finite extension of a field is algebraic; the finite extensions of Q are called the *algebraic number fields.* If k is an algebraic number field, it contains a set of elements which satisfy integer monic polynomials over Z; this is called the *ring of integers of* k, and is frequently denoted O_k.

In 1982, Lenstra, Lenstra, and Lovász [11] showed that $f(x)$ in $Q[x]$ can be factored in polynomial time. If $f(x)$ is a polynomial in O_k, where $k = Q[t]/g(t)$ is a number field, then there are polynomial time algorithms for factoring $f(x)$ over O_k [6], [10].

Following [3], a *formula* over a field k and its *depth of nesting* are defined as follows:

(1) an element of k is a formula of depth 0 over k,

(2) an arithmetic combination (A±B, A × B, A/B) of formulas A and B is a formula whose depth over k is max(depth(A), depth(B)), and

(3) a root $\sqrt[n]{A}$ of a formula A is a formula whose depth over k is 1+ depth(A).

We will call such a formula a nested radical. A nesting of α means any formula A that can take α as a value.[1] We will say the formula A *can be denested over the field* k if there is a formula B of lower depth than A such that A=B. We will say that *A can be denested in the field L* if there is a formula B=A of lower nesting depth than A with all of the terms (subexpressions) of B lying in L. For any α, we define the depth of α over k to be the depth of the minimum depth expression for α. When we are given a formula A for α such that A can be denested, we will sometimes instead say that α can be denested. We will write a primitive nth root of unity as a special symbol ζ_n rather than as a nested radical, and we will define the depth of nesting for a primitive root of unity that is not already in the field to be 1.

We return to the examples mentioned earlier. At first they seem mysterious. In fact, each equation is a result of a complex algebraic structure. Consider:

$$\sqrt[3]{\sqrt[3]{2}-1} = \sqrt[3]{1/9} - \sqrt[3]{2/9} + \sqrt[3]{4/9}.$$

A natural question to ask is: What is the relationship between $\sqrt[3]{2}$ and $\sqrt[3]{9}$? There is no obvious one. But there is a relationship between

$$\sqrt[3]{\sqrt[3]{2}-1}$$

and $\sqrt[3]{9}$; $\sqrt[3]{9}$ is an element of the field

$$Q(\sqrt[3]{\sqrt[3]{2}-1}).$$

This fact is unexpected.

The second and third equations in §1 give similar results. A natural question arises: In which field do denestings occur?

Let k be an algebraic number field, and let $f(x)$ be an irreducible polynomial of degree n with coefficients in k, and roots $\alpha_1, \cdots, \alpha_n$. Then $k(\alpha_i) \simeq k[x]/f(x) \simeq k(\alpha_j)$, but in general $k(\alpha_i) \neq k(\alpha_j)$. The field $L = k(\alpha_1, \cdots, \alpha_n)$ is called the *splitting field of $f(x)$ over k*. This is the smallest field containing $k(\alpha_1)$ that is Galois[2] over k. We consider the set of automorphisms of L which leave k fixed. These form a group, called the *Galois group of L* over k. As we can think of these automorphisms as permutations of the α_i, this group is sometimes called the *Galois group of $f(x)$ over* k. The Galois group is *transitive* on $\{\alpha_1, \cdots, \alpha_n\}$, that is, for each pair of elements α_i, α_j, there is an element σ in G, with $\sigma(\alpha_i) = \alpha_j$. Galois's great insight was the discovery of the relationship between the subgroups of G and the subfields of L containing k.

Let H be a subgroup of G. We denote by L^H the set of elements of L which are fixed pointwise by each element of H. This set forms a field. Furthermore, H fixes k, so that we have:

$$k \subseteq L^H \subseteq L.$$

[1] Of course, an nth root is a multivalued function. See §3 for a discussion on ambiguity.

[2] We say a field $F \supseteq k$ is Galois over k if every irreducible polynomial $p(x)$ in $k[x]$ which has a root in F splits completely in F.

Conversely, suppose that $J = k(\beta_1, \cdots, \beta_r)$ is a field such that $k \subset J \subset L$. Then the β_i can be written as polynomials in $\alpha_1, \cdots, \alpha_n$, and H, the subgroup of G which fixes J, consists of those elements of G which fix the β_i pointwise. The relationship between the fields and the groups can be formally stated as Theorem 1.1.

THEOREM 2.1 (Fundamental Theorem of Galois Theory). *Let k be a field, and let $f(x)$ in $k[x]$ be a polynomial of degree n, with roots $\alpha_1, \cdots \alpha_n$. Then:*

1. *Every intermediate field J, with $k \subset J \subset L = k(\alpha_1, \cdots, \alpha_n)$ defines a subgroup H of the Galois group G, namely, the set of automorphisms of L which leave J fixed. Furthermore, L is Galois over J.*
2. *The field J is uniquely determined by H, for J is the set of elements of L which are invariant under the action of H.*
3. *The subgroup H is normal if and only if J over k is a Galois extension. In that case the Galois group of J over k is G/H.*
4. *$|G| = [L : k]$, and $|H| = [L : J]$.*

The Galois groups we will be looking at are rather special. Our field extensions are extensions by radicals. Thus the groups we are looking at are solvable, that is, there is a sequence of subgroups $G = G_0 \supset G_1 \supset \cdots \supset G_t$, with G_{i+1} normal in G_i (written $G_{i+1} \lhd G_i$), and G_i/G_{i+1} cyclic of prime order. We introduce the following definiton.

DEFINITION. The commutator subgroup DG of G is the subgroup $\langle \sigma\tau\sigma^{-1}\tau^{-1} | \sigma, \tau \in G \rangle$. We will denote $D^2G = D(DG)$ and $D^iG = D(D^{i-1}G)$ for $i > 2$.

If G is a solvable group, there is an s such that $D^sG = \{e\}$. The sequence $D^iG/D^{i+1}G$ is called the derived series of G. The following are well known.

LEMMA 2.2. *Let $G = H_0 \supset H_1 \supset \cdots \supset H_t$ be a sequence of groups such that $H_i \lhd H_{i-1}$ and H_{i-1}/H_i is abelian. Then $H_i \supset D^iG$.*

LEMMA 2.3. *The groups D^iG are normal in G for all i.*

LEMMA 2.4. *If N is normal in G, then $D^i(G/N) \simeq D^i(G)N/N$.*

We say that an extension K over k is abelian if it is Galois and the resulting Galois group is abelian. Similarly, we say an extension is cyclic if it is Galois, and the resulting Galois group is cyclic. If $K = k(\beta)$ for some β that satisfies an irreducible polynomial of the $x^n - b$ for some b in k, we say that K is a simple radical extension of k. A tower of such extensions will be called a radical extension. Let ζ_n denote a primitive nth root of unity. The following classical theorems will prove useful.

THEOREM 2.5. *Let k be a field. The following are equivalent:*

1. *α is a nested radical over k.*
2. *There exists a solvable Galois extension L over k with α in L.*
3. *The splitting field of $k(\alpha)$ over k has solvable Galois group.*

This will be made more precise in Lemma 4.1, where the depth of nesting of a radical expression for α is related to the length of the derived series for the Galois group of L over k.

THEOREM 2.6. *Let k be a field, with K a cyclic extension of k of degree n, and suppose ζ_n is in k. Then there is a β in K such that $K = k(\beta)$, and β satisfies $x^n - b$ for some b in k.*

THEOREM 2.7 (Hilbert's Theorem 90). *Let K be a cyclic extension over k, and let σ be a generator of the Galois group G. For every element β in K with norm 1, there is an element $\gamma \neq 0$ in K such that $\beta = \gamma/\sigma(\gamma)$.*

THEOREM 2.8. *Let k be a field with ζ_n in k, and suppose β is a root of $x^n - b$. Then $k(\beta)$ is cyclic over k of degree d, where d divides n, and β^d is an element of k.*

If ζ_n is not in k, the situation is not quite as simple.

THEOREM 2.9. *Let k be a field, and n an integer greater than or equal to 2. Let a be an element of k, $a \neq 0$. Assume that for all prime numbers p dividing n, that a is not a pth power in k, and moreover, if $4 \mid n$, then a is not equal to $-4j^4$ for some j in k. Then $x^n - a$ is irreducible in $k[x]$.*

We are now ready to state our two main results which we will prove in §4.

THEOREM 4.2. *Suppose α is a nested radical over k, where k is a field of characteristic 0 containing all roots of unity. Then there is a minimal depth nesting of α with each of its terms lying in the splitting field of the minimal polynomial of α over k.*

THEOREM 4.7. *Suppose α is a nested radical over k, where k is a field of characteristic 0. Let L be the splitting field of $k(\alpha)$ over k, with Galois group G. Let l be the lcm of the exponents of the derived series of G. If there is a denesting of α such that each of the terms has depth no more than t, then there is a denesting of α over $k(\zeta_l)$ with each of the terms having depth no more than $t+1$ and lying in $L(\zeta_l)$.*

We also have an alternative version of this result in which we can achieve minimal depth at the expense of adjoining a primitive rth root of unity where r is dependent upon the presentation of the input.

COROLLARY 4.8. *Let k, α, L, G, l, t be as in Theorem 4.7. Let m be the lcm of the (m_{ij}), where the m_{ij} runs over all the roots appearing in the given nested expression for α. Let r be the lcm of (m, l). Then there is a minimal depth nesting of α over $k(\zeta_r)$ with each of its terms lying in $L(\zeta_r)$.*

Roots of unity, and indeed the radicals themselves, are more complicated than they might first appear. Before we prove the theorems, we discuss ambiguity of radical expressions and primitive roots of unity more closely.

3. Ambiguity. When we write the equation

$$\sqrt{5+2\sqrt{6}} = \sqrt{2}+\sqrt{3},$$

it is ambiguous. Which $\sqrt{2}$ do we mean? Which $\sqrt{3}$? The usual interpretation is the positive real roots for all four choices in the equation above. Under those choices, the equation is correct; under others, it may not be. Similarly, when we talk about denesting a nested radical, we must be careful about what we mean.

For example, suppose we are interested in denesting the expression:

$$\sqrt[3]{\sqrt[3]{2}-1} - \sqrt[3]{1/9}.$$

The polynomial $x^3 - 9$ factors over the field

$$Q(\sqrt[3]{\sqrt[3]{2}-1}).$$

To denest

$$\sqrt[3]{\sqrt[3]{2}-1} - \sqrt[3]{1/9},$$

we need to know to which root of $x^3 - 9$ we are referring in

$$Q(\sqrt[3]{\sqrt[3]{2}-1}),$$

the one which satisfies $x - \alpha^8 - 4\alpha^5 - 4\alpha^2$, or one of the two satisfying $x^2 + (\alpha^8 + 4\alpha^5 + 4\alpha^2)x + (3\alpha^4 + 6\alpha)$, where

$$\alpha = \sqrt[3]{\sqrt[3]{2} - 1}.$$

The problem, of course, is that $\sqrt[n]{\alpha}$ is a many-valued function. Once we choose which value it has, the same value must be assigned to it every time it appears. If the roots are specified at the time a nested radical is given, there is no difficulty, and we will choose those roots. If they are not, we have two choices: we could run the denesting algorithm for all possible choices of the values of the roots and compute the denesting in each case, or we could arbitrarily pick values. For the sake of simplicity, in this paper we choose to do the latter. When we adjoin $\sqrt[n]{\alpha}$, we do so in a way that makes the smallest (in terms of degree) field extension possible. In the above example, we would choose the $\sqrt[3]{9}$ that is already in the field.

Sometimes radicals may be given in a reducible form, e.g., $\sqrt[4]{4}$. The obvious simplification to $\sqrt{2}$ (which omits $-\sqrt{2}$ as per above) is not right, since the minimal polynomial for $\sqrt[4]{4}$ over Q has two roots $\pm i\sqrt{2}$, which this "simplification" omits.

Let $\alpha = \sqrt[n]{a}$ be a reducible radical, that is, suppose $x^n - a$ factors. By Theorem 2.9 this can happen in two ways. We might have a that is a pth power, where p divides n. In this case, $a = A^c$ where $gcd(c, n) = d$. Then the roots of $x^n - a$ are $\zeta_n^j \sqrt[n/d]{A^{c/d}}, j = 0, \cdots, n-1$. Note that $\sqrt[n/d]{A^{c/d}}$ is an irreducible radical. Thus what we can do is replace each instance of a reducible radical ($\sqrt[n]{A^c}$) by its irreducible cousin ($\sqrt[n/d]{A^{c/d}}$), and adjoin the appropriate ζ_n to the splitting field by letting the l of Theorem 4.7 be the lcm of the indices and the exponents of the derived series. The other situation, where $4 \mid n$ and $a = -4j^4$ for some j in the base field is handled similarly. Because of the way we handle roots of unity, this will not increase the depth of nesting.

Roots of unity present a more serious problem. We have chosen to write a primitive lth root of unity as ζ_l, rather than as a nested radical. The reason is that we are trying to write expressions in as simple a form as possible. In many situations ζ_l is a more meaningful expression than the nested radical it represents.

There are difficulties with this approach. A primitive lth root of unity may be of $\log l$ nesting depth. The symbol ζ_l hides that complexity. Adding roots of unity to k changes k in surprising ways. By the Kronecker–Weber Theorem, every abelian extension over Q can be embedded in an cyclotomic extension. Thus, we may have the ill fortune to be expressing $\sqrt[n]{\alpha}$ over $Q(\zeta_l)$, where $\sqrt[n]{\alpha}$ is an irrational number that happens to be in $Q(\zeta_l)$. Such is the case for $\sqrt{5}$ in the field $Q(\zeta_5)$. Thus, $\sqrt{5}$ will be represented as a polynomial in ζ_5, rather than as the more usual expression $\sqrt{5}$. This type of simplification may drop us a single level of nesting.

A more serious problem is that in writing a root of unity as ζ_l we are in some sense masking it. There are subtle ways in which we pay for that. For example, we can write

$$\sqrt{\sqrt{5} - 5/2}$$

as $\zeta_5 - 1/\zeta_5$. Which symbol is easier to understand:

$$\sqrt{\sqrt{5} - 5/2} \quad \text{or} \quad \zeta_5 - 1/\zeta_5?$$

That is certainly open to debate. In choosing to express

$$\sqrt{\sqrt{5}-5/2}$$

as $\zeta_5 - 1/\zeta_5$, it is not clear that we have really simplified things. This problem seems unavoidable in the current approach.

If $k = Q$, then the minimal polynomial of ζ_l is the unique irreducible factor of $x^l - 1$ of degree $\varphi(l)$. If $k \neq Q$, this minimal polynomial may factor. Since the primitive lth roots of unity are all powers of one another, the fields $k(\zeta_l)$ are all equal. The denested expression we find for α is dependent on our choice of ζ_l, however, a different ζ_l will give a different denesting expression. When we denest a nested radical α, we will always specify the minimal polynomial of the associated primitive lth root of unity.

Now we are in a position to prove our main theorems, which we will do in the next section.

4. Algebraic structure. In this section we discuss the algebraic structure surrounding nested radicals. We begin with Lemma 4.1.

LEMMA 4.1. *Let α be a nested radical, and suppose α can be denested in a field $L \supseteq k$. Suppose that the denested expression has nesting depth l. Let $\tilde{L}$ be the splitting field*[3] *of L over k, with Galois group G, and assume that all roots of unity of $\tilde{L}$ appear in k. Then there are subgroups $H_1, \cdots, H_l$ of G, with $G = H_0 \rhd H_1 \rhd \cdots \rhd H_l$ and H_i/H_{i+1} abelian for $i = 0, \cdots, l-1$, with $H \supseteq H_l$, where H is the subgroup of G associated with $k(\alpha)$. Conversely, if there is such a sequence of subgroups, then α has nesting depth at most l.*

Proof. If α can be denested, then $\alpha = q(\beta_1, \cdots, \beta_t)$, with each β_i having nesting depth $l_i \leq l$ over k. Each β_i has a nesting sequence of the form:

$$\begin{aligned}
\beta_{i1} &= \sqrt[m_{i1}]{p_{i1}}, \ p_{i1} \in k, \\
\beta_{i2} &= \sqrt[m_{i2}]{p_{i2}(\beta_{11}, \cdots, \beta_{t1})}, \\
\beta_{i3} &= \sqrt[m_{i3}]{p_{i3}(\beta_{11}, \cdots, \beta_{t1}, \beta_{12}, \cdots, \beta_{t2})}, \\
&\vdots \\
\beta_i = \beta_{il_i} &= \sqrt[m_{il_i}]{p_{il_i}(\beta_{11}, \cdots, \beta_{tl_t-1})},
\end{aligned}$$

where the p_{ij}'s and q are multivariate polynomials over k. (Since each β_i has a possibly different depth of nesting, without loss of generality, let $l_t = l$ be the maximal depth of nesting, and let $\beta_{ij} = \beta_i$ for $j > l_i$.) Without loss of generality assume that

$$\sqrt[m_{ij}]{p_{ij}(\beta_{11}, \cdots, \beta_{tj-1})}$$

is of degree m_{ij} over $k(\beta_{11}, \cdots, \beta_{tj-1})$. Since $x^{m_{ij}} - p_{ij}(\beta_{11}, \cdots, \beta_{tj-1})$ has a root in $\tilde{L}$, it splits completely in $\tilde{L}$. Thus we know that $\zeta_{m_{ij}}$ is in k for every pair i, j. (Since all the necessary roots of unity lie in k, we do not run into problems with ambiguity.)

Number the extensions of k in the following way: $k_1 = k(\beta_{11}, \cdots, \beta_{t1}) \subseteq k_2 = k_1(\beta_{12}, \cdots, \beta_{t2}) \subseteq \cdots \subseteq k_l = k_{l-1}(\beta_{1l}, \cdots, \beta_{tl}) = k(\beta_{11}, \cdots, \beta_{tl})$. Let H_i be the subgroup of G corresponding to k_i. Observe that k_{i+1} is a Galois extension of k_i,

[3] This is usually called the normal closure of L over k, i.e., the minimal field $\tilde{L}$ containing L in which every polynomial in $k[x]$ that has a root in L splits completely in $\tilde{L}$.

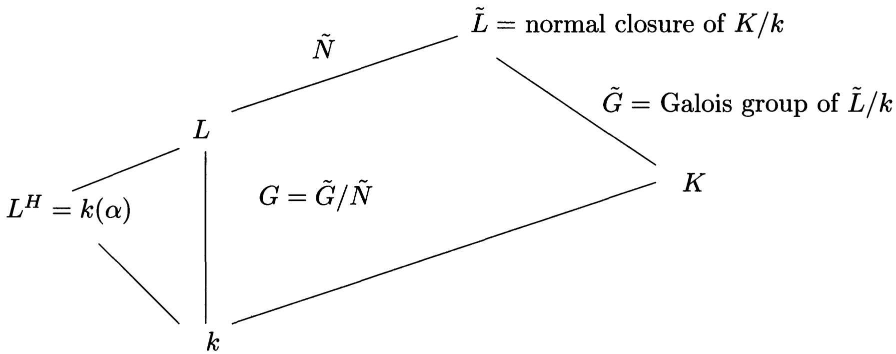
since the appropriate roots of unity lie in k. Thus we have $G = H_0 \triangleright H_1 \triangleright \cdots \triangleright H_l$. Furthermore, k_{i+1} is an abelian extension of k_i, since it is contained in the composite of cyclic extensions. Thus H_i/H_{i+1} is abelian. Finally, $k(\alpha) \subset k(\beta_{11}, \cdots, \beta_{tl_t})$; thus $H \supset H_l$.

The converse follows immediately from Theorems 2.5 and 2.6 and the fact that the needed roots of unity lie in k. □

We are now ready to proceed with Theorem 4.2

THEOREM 4.2. *Suppose α is a nested radical over k, a field of characteristic 0 that contains all roots of unity. Then there is a minimal depth nesting of α with each of its terms lying in the splitting field of the minimal polynomial of α over k.*

Proof. Let L be the splitting field of $k(\alpha)$ over k, with Galois group G. Suppose α can be denested over k with all of its terms lying in K, a field. Let $\tilde{L}$ be the normal closure of K over k; then $\tilde{L} \supset L$. Let $\tilde{G}$ be the Galois group of $\tilde{L}$ over k; then the field $\tilde{L}$ is a Galois extension over L with group $\tilde{N}$. Furthermore, by the Fundamental Theorem of Galois Theory, $G \simeq \tilde{G}/\tilde{N}$. Let H be the subgroup of G associated with $k(\alpha)$, and let $\tilde{H}$ be the pullback of H. Then $\tilde{H} \supset \tilde{N}$.



Now since α can be denested over $\tilde{L}$, by Lemma 4.1 there is a series of subgroups $\tilde{H}_1, \cdots, \tilde{H}_l$ of $\tilde{G}$ such that $\tilde{G} = \tilde{H}_0 \triangleright \tilde{H}_1 \triangleright \cdots \triangleright \tilde{H}_l$, with $\tilde{H}_i/\tilde{H}_{i+1}$ abelian for $i = 0, \cdots, l-1$, and $\tilde{H} \supset \tilde{H}_l$. We will show how to pull this down to a sequence in G, thus showing that α can be denested in L.

Consider the sequence $\tilde{H}_1\tilde{N}, \tilde{H}_2\tilde{N}, \cdots, \tilde{H}_l\tilde{N}$. Clearly, $\tilde{H}_i\tilde{N} \triangleright \tilde{H}_{i+1}\tilde{N}$. That $\tilde{H}_i\tilde{N}/\tilde{H}_{i+1}\tilde{N}$ is abelian follows immediately from the facts that (i) $\tilde{H}_i/\tilde{H}_{i+1}$ is abelian, (ii) $\tilde{N}$ is normal in $\tilde{G}$. Let $a\tilde{H}_{i+1}\tilde{N}, b\tilde{H}_{i+1}\tilde{N}$ be elements of $\tilde{H}_i\tilde{N}/\tilde{H}_{i+1}\tilde{N}$. Then there are a_1, a_2, b_1, b_2 in $\tilde{H}_i$ and n_1, n_2, n_3, n_4 in $\tilde{N}$ such that $a = n_1a_1 = a_2n_2$ and $b = n_3b_1 = b_2n_4$. Then $a\tilde{H}_{i+1}\tilde{N}b\tilde{H}_{i+1}\tilde{N} = b\tilde{H}_{i+1}\tilde{N}a\tilde{H}_{i+1}\tilde{N}$.

Finally, since $\tilde{H} \supset \tilde{N}$ and $\tilde{H} \supset \tilde{H}_l$, we know that $\tilde{H} \supset \tilde{H}_l\tilde{N}$.

Now consider the sequence H_i in G defined by $\tilde{H}_i\tilde{N}/\tilde{N} = H_i$. That the sequence $G = H_0, H_1, \cdots, H_l$ satisfy $H_i \triangleright H_{i+1}$ for $i = 0, \cdots, l-1$ follows from the third isomorphism theorem. That H_i/H_{i+1} is abelian follows from the fact that

$$H_i/H_{i+1} = \frac{\tilde{H}_i\tilde{N}/\tilde{N}}{\tilde{H}_{i+1}\tilde{N}/\tilde{N}} \simeq \tilde{H}_i\tilde{N}/\tilde{H}_{i+1}\tilde{N}.$$

The isomorphism is a consequence of the third isomorphism theorem. That $H \supset H_l$ is clear, since $H = \tilde{H}\tilde{N}/\tilde{N} \supset \tilde{H}_l\tilde{N}/\tilde{N} = H_l$.

Thus we have taken a chain of subgroups in $\tilde{G}$ and mapped it to one over G. The fact that all the needed roots of unity lie in k means that we can apply Theorem 2.8,

and we have taken a denesting over $\tilde{L}$ and mapped it to one over L. The theorem is proved. □

This is a pleasing theorem. Although it is subsumed by Theorem 4.7, its proof is different, and it shows how any denesting expression for α can be mapped to one in which all the subexpressions are in L, the splitting field of α over k, assuming all roots of unity lie in k. By Lemma 2.2, a sequence of minimal nesting depth for α is found by finding the derived series and computing the associated fields. This will be explained in detail in §5.

In general, it will not be the case that all roots of unity lie in k. Of course, one can adjoin them, but this may be an infinite extension. From a computational standpoint this is not a good approach. Instead we will find a single primitive root of unity we can adjoin to k that will give us a denesting. We begin with the following definition.

DEFINITION. The least positive integer n such that $G^n = \{e\}$ is the exponent of G.

THEOREM 4.3. *Suppose $k \subset k_1 \subset \cdots \subset k_t$ is a series of abelian extensions (k_{i+1} an abelian extension of k_i), with $K = k(\alpha) \subset k_t$. Let L be the splitting field of K over k, and L_1 be the splitting field of k_t over k, with Galois group G. Then $L \subset L_1$, and let $N = Gal(L_1/L)$. If s is minimal such that $D^sG \subset N$, then $s \leq t$. Furthermore s is the length of the derived series for $G/N = Gal(L/k)$.*

Proof. We begin with some notation. Let

$$\begin{aligned} Gal(L_1/K) &= H_0, \\ Gal(L_1/k) &= G, \\ Gal(L_1/k_i) &= H_i, i = 1, \cdots, t, \\ Gal(L_1/k_t) &= H_t = H. \end{aligned}$$

Observe that $G \supset H_1 \supset \cdots \supset H_t = H$, and that $H_i \supset D^iG$, since H_{i-1}/H_i is abelian. Furthermore, we know that $\bigcap_{\sigma \in G} \sigma H \sigma^{-1} = \{e\}$, since L_1 is the smallest normal extension of k_t over k. Similarly, $\bigcap_{\sigma \in G} \sigma H_0 \sigma^{-1} = N$, since L is the smallest normal extension of K over k, and $N = Gal(L_1/L)$. Note that N is a normal subgroup of G.

Now let s be the least integer such that $D^sG \subset N$. We know that $\{e\} = \bigcap_{\sigma \in G} \sigma H_t \sigma^{-1} \supset \bigcap_{\sigma \in G} \sigma D^tG \sigma^{-1} = \bigcap D^tG = D^tG$. Thus $D^tG = \{e\}$. This implies that $D^tG \subset D^sG$, or that $s \leq t$. Furthermore, note that $D^i(G/N) \simeq D^i(G)N/N$, which implies that $D^s(G/N) = \{e\}$. Since s is the least integer such that $D^sG \subset N$, we have that s is the length of the derived sequence of $G/N = Gal(L/k)$. □

This theorem almost gives us a minimal denesting. Let l be the lcm of the exponents of the derived series of $G = Gal(L/K)$. Then if ζ_l, a primitive lth root of unity, is in k, by Theorems 2.5 and 2.8 all the extensions by the derived series can be achieved as radical extensions. If ζ_l is not in k, but is in L, we can still achieve a minimal denesting with all of the terms lying in L. This is because of the following lemma.

LEMMA 4.4. *Let L be the splitting field of the minimal polynomial of α over k, with Galois group G, and suppose ζ_l is in L. If $H = DG$ is the commutator subgroup of G, then $L^H \supseteq k(\zeta_l)$.*

Proof. Let J be the subgroup of G associated with $k(\zeta_l)$. Now $k(\zeta_l)$ is an abelian extension of k, thus $J \lhd G$, and G/J is abelian. By Lemma 2.2, $J \supseteq DG$, and therefore $k(\zeta_l) = L^J \subseteq L^{DG}$. □

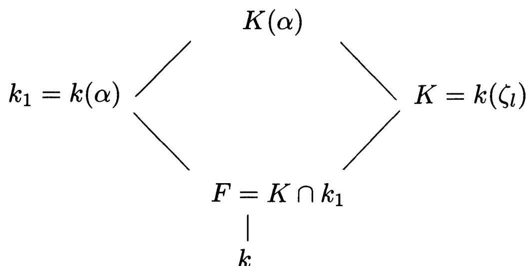
Of course, it will not always be the case that ζ_l lies in L. We can adjoin it to k.

Thus the field extensions of k made by considering the fixed fields L^{D^iG} can all be made to be radical extensions. We will need to make use of Theorem 4.5.

THEOREM 4.5 (Lang [8, pp. 196–197]). *Let K be a Galois extension of k, and F be an arbitrary extension of k. Then KF is Galois over F, and K is Galois over $K \cap F$. Let H be the Galois group of KF over F, and G the group of K over $K \cap F$. If σ is in H, then the restriction of σ to K is in G, and the mapping $\sigma \to \sigma|_K$ is an isomorphism.*

LEMMA 4.6. *Suppose $k_1 = k(\alpha)$ is an abelian extension of k with group G, which has exponent l. Let ζ_l be a primitive lth root of unity, and let $K = k(\zeta_l)$. Then $K(\alpha)$ is an abelian extension of K, and if $\tilde{G}$ is the associated Galois group, l is divisible by the exponent of $\tilde{G}$. If, furthermore, $k_1 = k(\alpha)$ is a cyclic extension of k, then $K(\alpha)$ is a radical extension of K, that is, $K(\alpha) = K(\beta)$, for some β which satisfies an irreducible polynomial of the form $x^n - b$ for some b in K.*

Proof. Consider the following picture:



Now k_1 is an abelian extension over k; thus so is k_1 over F. Furthermore, by Theorem 4.5, $\tilde{G}$ is isomorphic to $Gal(k_1/F)$, which is just a subgroup of G. Thus the exponent of $\tilde{G}$ divides the exponent of G. If k_1 is a cyclic extension over k, so is k_1 over F. Thus so is $K(\alpha)$ over K. Again $\tilde{G}$ can be viewed as a subgroup of G. The exponent of $\tilde{G}$ will divide the exponent of G. But then, by Theorem 2.6, the extension $K(\alpha)$ over K can be realized as a radical extension. □

THEOREM 4.7. *Suppose α is a nested radical over k, where k is a field of characteristic 0. Let L be the splitting field of $k(\alpha)$ over k, with Galois group G. Let l be the lcm of the exponents of the derived series of G. If there is a denesting of α such that each of the terms has depth no more than t, then there is a denesting of α over $k(\zeta_l)$ with each of the terms having depth no more than $t+1$, and lying in $L(\zeta_l)$.*

Proof. Since α is a nested radical with nesting depth t, α can be expressed as a polynomial in $\beta_1, \cdots, \beta_r$, with each β_i having a nesting depth $t_i \le t$ over k. Each β_i has a nesting sequence of the form:

$$\begin{aligned}
\beta_{i1} &= \sqrt[m_{i1}]{p_{i1}}, \ p_{i1} \in k, \\
\beta_{i2} &= \sqrt[m_{i2}]{p_{i2}(\beta_{11}, \cdots, \beta_{r1})}, \\
\beta_{i3} &= \sqrt[m_{i3}]{p_{i3}(\beta_{11}, \cdots, \beta_{r1}, \beta_{12}, \cdots, \beta_{r2})}, \\
&\vdots \\
\beta_i = \beta_{it_i} &= \sqrt[m_{il_i}]{p_{il_i}(\beta_{11}, \cdots, \beta_{rt_{r-1}})}.
\end{aligned}$$

Of course, the symbol $\sqrt[m_{ij}]{p_{ij}(\beta_{11},\cdots,\beta_{rj-1})}$ will mean the same thing each time it appears. Let $m = lcm(m_{11},\cdots,m_{1l_1},\cdots,m_{r1},\cdots,m_{rl_r})$, and let $\hat{k} = k(\zeta_m)$.

Consider the sequence of fields $k_1 = \hat{k}(\beta_{11},\cdots,\beta_{1r}) \subseteq k_2 = k_1(\beta_{21},\cdots,\beta_{2r}) \subseteq \cdots \subseteq k_t = k_{t-1}(\beta_{1t},\cdots,\beta_{rt})$. For each i, k_{i+1} is a composite of cyclic extensions of k_i. Thus for each i, k_{i+1} is an abelian extension of k_i. Let L be the splitting field of $k(\alpha)$ over k, with Galois group G. We know that L is contained in the normal closure of k_t over k. If the length of the derived series for G is s, then by Theorem 4.3, $s \leq t+1$.

Let $L_i = L^{D^iG}$. Then the sequence of fields $k = L_0 \subset L_1 \cdots \subset L_{s-1} \subset L_s = L$ is a tower of abelian extensions. The associated Galois group is $D^{i-1}G/D^iG$.

Each $D^{i-1}G/D^iG$ is an abelian group and can be written as a direct product of cyclic groups, say, $D^{i-1}G/D^iG = J_{i1} \times \cdots \times J_{it_i}$. For each i and j, $i = 1,\cdots,s$, $j = 1,\cdots,t_i$, let $\tilde{J}_{ij} = \{e\} \times \cdots \{e\} \times J_{ij} \times \{e\} \cdots \{e\} \subseteq D^iG/D^{i-1}G$. Then $L_i = L^{\tilde{J}_{ij}}$ is a cyclic extension of L_{i-1}, and

$$L_i = L_i^{\tilde{J}_{i1}} L_i^{\tilde{J}_{i2}} \cdots L_i^{\tilde{J}_{it_i}}$$

is a composite of cyclic extensions.

Now let $K_i = L_i(\zeta_l)$. By Lemma 4.6 the extension K_i over K_{i-1} is abelian, and the extension

$$K_{ij} = L_i^{\tilde{J}_{ij}}(\zeta_l)$$

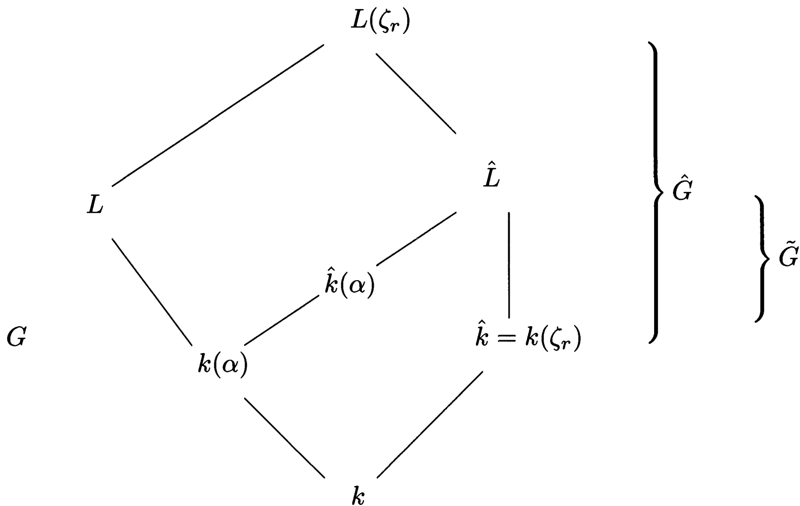
over K_i is a radical extension for every pair i,j. The maximal depth of nesting for any expression in K_s over k is s, the height of the derived series for G. The theorem is proved. □

Observe that the reason the depth of nesting achieved in Theorem 4.7 is one more than minimal is because the abelian tower created by the derived series begins with the field k rather than $k(\zeta_m)$. But ζ_m is, after all, just a root of unity. If we begin our tower of fields at $k(\zeta_m)$ instead, then we have Corollary 4.8.

COROLLARY 4.8. *Let k,α,L,G,l,t be as in Theorem 4.7. Let m be the lcm of (m_{ij}), where the m_{ij} runs over all the roots appearing in the given depth t nested expression for α. Let r be the lcm of (m,l). Then there is a minimal depth nesting of α over $k(\zeta_r)$ with each of its terms lying in $L(\zeta_r)$.*

Proof. This proof is a variation on the one above. We want to show that there is a denesting of α over $k(\zeta_r)$ of minimal depth, with all of its terms lying in $L(\zeta_r)$. We do it by showing something slightly stronger, that there is a denesting achieving this with all of its terms lying in $\hat{L}$, the splitting field of the minimal polynomial of α over $k(\zeta_r)$.

We begin with some notation. Let $\hat{k} = k(\zeta_r)$, and let $\hat{G}$ be the Galois group of the splitting field of the minimal polynomial of α over $\hat{k}$. Note that by Theorem 4.5, $L(\zeta_r)$ over $\hat{k}$ is Galois. If $\tilde{G}$ is the group of that extension, then $\tilde{G}$ is isomorphic to a subgroup of G, the subgroup corresponding to L over $k(\alpha) \cap k(\zeta_r)$. (Note that $k(\alpha) \cap k(\zeta_r)$ is an abelian extension of k.) We have the following picture:



Consider the tower $\hat{k} \subset \cdots \hat{k}_t$, and let us compare it to a tower of abelian extensions in $\hat{L}$ over $\hat{k}$. Theorem 4.3 applies. If s be the length of the derived series for $\hat{G}$, we have $s \leq t$.

We let $\hat{L}_i = \hat{L}^{D^i\hat{G}}$. As in the previous theorem, the sequence of fields $\hat{k} \subset \hat{L}_1 \subset \cdots \subset \hat{L}_s$ is an abelian tower. Again, this can be transformed so that for each i, $\hat{L}_i$ is a composite of cyclic extensions of $\hat{L}_{i-1}$.

Since $\hat{G}$ is a quotient group of $\tilde{G}$, the lcm of the exponents of the derived series for $\hat{G}$ divides the lcm of the exponents for the derived series of $\tilde{G}$, which in turn divides the lcm of the exponents of the derived series for G. This divides r. Thus $\hat{k}$ contains the roots of unity needed to make the field extensions of $\hat{L}$ over $\hat{k}$ corresponding to the derived series into composites of radical extensions. The corollary follows. □

Although Corollary 4.8 appears to give a better bound than Theorem 4.7, it does so in a dissatisfying way, by introducing roots of unity which have to do with the input and are not necessarily a genuine part of the problem. For example, in denesting

$$\sqrt[3]{\sqrt{5}+2} - \sqrt[3]{\sqrt{5}-2}$$

over Q, Corollary 4.8 indicates that one would want to add a primitive sixth root of unity to Q to denest. In fact,

$$\sqrt[3]{\sqrt{5}+2} - \sqrt[3]{\sqrt{5}-2} = 1,$$

and thus no new root of unity is needed (a fact which would be discovered in computing the minimal polynomial for

$$\sqrt[3]{\sqrt{5}+2} - \sqrt[3]{\sqrt{5}-2}$$

over Q). For this reason we have chosen to use the Theorem 4.7 variety of denesting as the basis for our algorithms. The algorithms can be easily modified to handle the Corollary 4.8 version if desired.

Now we are in a position to denest general radicals. Theorem 4.7 tells us which root of unity we must add. Before we proceed with the algorithm, we briefly discuss computing the minimal polynomial of α; details may be found in the Appendix.

From an algebraic point of view, a sensible measure of the size of the denesting problem is the size of the minimal polynomial of α. There is some disagreement about this issue. Borodin et al. [3] define the size of α to be m, the depth of nesting of α. Under this measure, they gave an algorithm for denesting a nested class of square roots. This algorithm has polynomial arithmetical complexity, but exponential bit complexity. The latter comes from the doubling of bits which occurs at each extension. We believe that this problem is inherent in any denesting scheme, and that the measure defined by [3] is too conservative. Under our measure, their running time is polynomial.

We can view a nested radical α as a sequence of polynomials $\tilde{p}_i, \tilde{q}$ in $k[x_1, \cdots, x_{m-1}]$ such that

$$\begin{aligned}
\alpha_1 &= \sqrt[n_1]{\tilde{p}_1}, \ \tilde{p}_1 \in k, \\
\alpha_2 &= \sqrt[n_2]{\tilde{p}_2(\alpha_1)}, \\
\alpha_3 &= \sqrt[n_3]{\tilde{p}_3(\alpha_1, \alpha_2)}, \\
&\ \vdots \\
\alpha_m &= \tilde{q}(\alpha_1, \cdots, \alpha_{m-1}) + \sqrt[n_m]{\tilde{p}_m(\alpha_1, \cdots, \alpha_{m-1})},
\end{aligned}$$

with $\alpha = \alpha_m$. In the Appendix we show how to go from this description to minimal polynomial of $\alpha = \alpha_m$ over k.

5. The algorithm and running time analysis. At this point we have presented all the new ideas that were needed in order to compute denestings. In this section we show that we can effectively compute the denesting, and we give a running time analysis. There are some computations we have not yet shown how to do, the most important of which is how to go from an arbitrary description of a radical extension (e.g., $K = k[x]/f(x)$, $f(x)$ irreducible) to a description as a radical extension ($K = k[x]/(x^n - b)$, with $x^n - b$ irreducible). The answer is Lagrange resolvents, and we use these and Artin's normal basis to achieve what we want.

We begin this section with a brief—but complete—description of the algorithm. The rest of the section is concerned with the details of running time analysis and coefficient blowup. Because it is remarkably easy to miss the forest for the trees, we urge the first-time reader to read the brief—but complete—description of the algorithm, and then to read about normal bases and Lagrange resolvents. A second reading can cover the running time analyses, and the other algorithms presented. We end the section with a more precise version of the algorithm and, of course, a running time analysis.

Suppose we wish to denest the nested radical α. We begin by computing the minimal polynomial of α over k. We construct the splitting field L of the minimal polynomial of α over k. We compute $G = Gal(L/k)$, and the series of commutator subgroups $D^iG, i = 1, \cdots, s$, where $D^sG = \{e\}$. We also compute l, the lcm of the exponents of the derived series of G.

Next, for each $i, i = 1, \cdots, s$, we compute $D^{i-1}G/D^iG = J_{i1} \times \cdots \times J_{it_i}$ as a direct product of cyclic groups. Let $\tilde{J}_{ij} = \{e\} \times \cdots \times \{e\} \times J_{ij} \times \{e\} \times \cdots \times \{e\}$, and let $L_i = L^{D^iG}$. Thus for each i,

$$L_i = L_i^{\tilde{J}_{i1}} \cdots L_i^{\tilde{J}_{it_i}}$$

is a composite of cyclic extensions of L_{i-1}. For each i and j, we compute $\tilde{\beta}_{ij}$ such

that

$$L_i^{\tilde{J}_{ij}} = L_{i-1}(\tilde{\beta}_{ij}).$$

Thus $L_i = L_{i-1}(\tilde{\beta}_{i1}, \cdots, \tilde{\beta}_{it_i})$.

We write $K_0 = k(\zeta_l)$, where ζ_l is a primitive lth root of unity. By Lemma 4.6, $K_{ij} = K_{i-1}(\tilde{\beta}_{ij})$ can be written as a radical extension of K_{i-1}, and each $K_i = K_{i1} \cdots K_{it_i}$ is a composite of radical extensions of K_{i-1}. We achieve the radical extensions as follows.

Following Artin, we construct a polynomial $s_{ij}(x)$ whose roots $\theta_{ij1}, \cdots, \theta_{ijr_{ij}}$ form a "normal" basis for K_{ij} over K_{i-1}. The degree of $s_{ij}(x)$ is $r_{ij} = [K_{ij} : K_{i-1}]$, and its roots are linearly independent over K_{i-1}. Then we will use Lagrange resolvents to find a β_{ij} in K_{ij} such that $K_{ij} = K_{i-1}(\beta_{ij})$, where β_{ij} satisfies an irreducible polynomial of the form $x^{n_{ij}} - b_{ij}$ over K_{i-1}. In our construction we will actually have two descriptions of the fields K_i, namely, $K_{i-1}(\beta_{i1}, \cdots, \beta_{it_i})$ and $K_{i-1}[x]/g_i(x)$, where $g_i(x)$ is irreducible. We will do computations using the latter representation.

In this section we show that all the computations described above can be carried out in time polynomial in the size of the splitting field of the minimal polynomial of α over k. Unfortunately, the splitting field can be large. In worst case, the splitting field is of exponential degree over the base field. Our bounds are chosen for the relative simplicity of presentation, and are not necessarily the tightest possible. We will restrict our running time analysis to $k = Q$ for the present.

In [6], it was observed that the splitting field and Galois group G can be computed in time polynomial in $r = |G| = [L : k]$ and $\log |f(x)|$. More precisely, we have Theorem 5.1.

THEOREM 5.1. *Let $f(x)$, an irreducible polynomial of degree m over k, be the minimal polynomial of α, a nested radical over k. The Galois group of $f(x)$ over k can be computed in $O(r^{12+\epsilon} \log^{4+\epsilon}(r|f(y)|))$ steps.*

Proof. Suppose $f(x)$ has roots $\alpha_1, \cdots, \alpha_m$. The following algorithm computes L, the splitting field of $f(x)$ over k, and G, the associated Galois group.

Step 1: $g(x) \leftarrow f(x)$;
Step 2: WHILE $f(y)$ does not split completely in $k[x, y]/g(x)$ DO:
  BEGIN
    Factor $f(y) = \Pi f_i(y)$ in $k[x, y]/g(x)$;
    Let $h(y)$ be some $f_i(y)$ of degree > 1;
    Pick c such $g(x) \leftarrow Res_t(g(t), h(x - ct))$ is square-free;
  END;
Step 3: $\rho \leftarrow$ root of $g(x)$;
  Factor $g(y) = \Pi(y - p_i(\rho))$ in $k[x, y]/g(x)$;
  $\sigma_i(\rho) \leftarrow p_i(\rho)$;
  FOR each σ_i in G, compute σ_i acting on ρ_j by
  $\sigma_i(\rho_j) = \sigma_i(p_j(\rho)) = p_j(\sigma_i(\rho)) = p_j(p_i(\rho))$.

By Theorem A.4, it is not hard to see that this algorithm computes a polynomial $g(x)$ such that $L = k[x]/g(x)$. The roots of this polynomial are the conjugates of

$$\gamma_1 = (c_1 \cdots c_{m-1})^2 \alpha_1 + (c_1 \cdots c_{m-2})^2 \alpha_2 + \cdots + \alpha_m,$$

where

$$c_i < d_i = [k(\alpha_1, \cdots, \alpha_i) : k(\alpha_1, \cdots, \alpha_{i-1})].$$

How long the process takes depends on the structure of G. We will assume a worst case scenario. Note that not all worst case assumptions can occur simultaneously (and thus what we have is really an exaggerated worst case scenario).

We loop through Step 2 at most $m \leq \log r$ times. The maximal possible $g(x)$ occurs in the last iteration of the loop, and it is bounded by the size of its roots. Now $g(x)$ is of degree r over k in the final iteration. Then

$$|g(x)| \leq r\binom{r}{r/2}||\gamma_1||^r < 2^r(r^2||\alpha_1||)^r < (r||\alpha_1||)^{2r} < (r|f(y)|)^{2r+\epsilon},$$

since $||\alpha_1|| < 1 + |f(y)|$. Using Theorem A.4 and the fact that degree$(g(x))\times$ degree$(f(x)) = r$, we have that factoring $f(y)$ over $k[x]/g(x)$ takes no more than

$$O((r/m)^{9+\epsilon}m^{7+\epsilon}\log^2(g(x))\log^{2+\epsilon}(||f(y)||(r|g(x)|)^r(rm)^m))$$

steps. We loop through at most m times, getting

$$\begin{aligned} &O((r/m)^{9+\epsilon}m^{7+\epsilon}\log^2(r|f(y)|)^{2r+\epsilon}\log^{2+\epsilon}(|f(y)|(r(r|f(y)|)^{2r+\epsilon})^r(rm)^r)m) \\ &< O(r^{9+\epsilon}\log^2(r|f(y)|)^{2r+\epsilon}\log^{2+\epsilon}(r|f(y)|)^{4r^2}) \\ &< O(r^{12+\epsilon}\log^{4+\epsilon}(r|f(y)|)) \end{aligned}$$

steps. □

Thus in time polynomial in the degree of the splitting field, and the size of the minimal polynomial of α over k, we can compute the Galois group. It has long been known how to compute commutator subgroups quickly from a group table; recently Babai, Luks, and Seress [2] showed how to do so in $O(r^4\log^c r)$ steps, for a permutation group on r elements.

We say a polynomial $g(x)$ over a field k is normal if and only if it is irreducible and $k[x]/g(x)$ is a Galois extension of k (equivalently $g(y)$ splits completely in $k[x]/g(x)$). It is not hard to go from such a polynomial to one which splits completely and whose roots are linearly independent over k. Artin gave an effective method for calculating such a polynomial. We reproduce it here.

THEOREM 5.2 (Artin [1]). *If L is a Galois extension of k, and $\sigma_1, \cdots, \sigma_r$ are the elements of the group G, then there is an element θ in L such that the r elements $\theta_1 = \sigma_1(\theta), \cdots, \theta_r = \sigma_r(\theta)$ are linearly independent with respect to k.*

Proof. Let $L = k(\rho)$, and let $g(x)$ be the minimal polynomial for ρ over k. Let $\sigma_i(\rho) = \rho_i$. We let $h(x) = g(x)/(x-\rho)g'(\rho)$, and $h_i(x) = \sigma_i(h(x)) = g(x)/(x-\rho_i)g'(\rho_i)$. Then $h_j(x)$ is a polynomial over L having ρ_i as a root for $j \neq i$, and thus $h_i(x)h_j(x) = 0 \pmod{g(x)}$ for $i \neq j$. In the equation

$$h_1(x) + h_2(x) + \cdots + h_r(x) - 1 = 0, \tag{1}$$

the left side is of degree at most $r-1$. If the equation were true for r different values of x, the left side would be identically 0. But we have r such values $\rho_1, \cdots, \rho_r$, since $h_i(\rho_i) = 1$, and $h_j(\rho_i) = 0$ for $j \neq i$. Multiplying this equation by $h_i(x)$, and using the fact that $h_i(x)h_j(x) = 0 \pmod{g(x)}$ for $i \neq j$, we find that

$$(h_i(x))^2 = h_i(x) \pmod{g(x)}. \tag{2}$$

We next consider the determinant

$$D(x) = |\ \sigma_i\sigma_j(h(x))\ |, \tag{3}$$

and prove that $D(x)$ is not identically 0. If we square the corresponding matrix we get the identity matrix (mod$g(x)$), because of (1)–(3).

Now $D(x)$ can have at most r^2 roots in k. If we avoid them, we can find a value a for x such that $D(a) \neq 0$. Let $\theta = h(a)$. Then

$$| \sigma_i \sigma_j(\theta) | \neq 0.$$

Consider any relation of the form $a_1\sigma_1(\theta) + \cdots + a_r\sigma_r(\theta) = 0$, where the a_i are in k. Applying the automorphisms σ_j to it would lead to r homogeneous equations for the r unknowns a_i. Then $a_i = 0$, and we are done. □

ALGORITHM: COMPUTE NORMAL BASIS.
input: $g(x)$, a normal polynomial over k with root ρ
output: θ, an element in $k[x]/g(x)$ whose conjugates form a normal basis for $k[x]/g(x)$ over k.

Line 1: BEGIN
Line 2: $h(x) \leftarrow g(x)/(x-\rho)g'(\rho)$;
Line 3: FOR $i = 1, \cdots, r$ DO:
FOR $j = 1, \cdots, r$ $a_{ij} \leftarrow \sigma_i\sigma_j(h(x))$;
Line 4: $D(x) \leftarrow | a_{ij} |$;
Line 5: Pick a in k such that $D(a) \neq 0$;
Line 6: $\theta \leftarrow h(a)$;
Line 7: END.

THEOREM 5.3. *Algorithm Compute Normal Basis computes a normal polynomial whose roots form a basis for $k[x]/g(x)$ over k. It does so in $O(r^{7+\epsilon}\log^{2+\epsilon}|f(y)|)$ steps.*

Proof. That it does so correctly is clear from the proof of Artin's theorem. We already know by Theorem 5.1 that

$$|g(x)| < (r|f(y)|)^{2r+\epsilon}.$$

Thus

$$|g'(x)| < (r|f(y)|)^{2r+\epsilon'}$$

and

$$|h(x)| < (r|f(y)|)^{4r+\epsilon}.$$

We know the entries in the matrix are bounded by $|h(x)|$, and that $D(x)$ is a polynomial of degree at most r^2. By Hadamard's inequality [5], the coefficients of $D(x)$ are bounded by $(r|f(y)|)^{9r^2}$. There is an a less than r^2 for which $D(a)$ is nonzero. Now

$$\theta = h(a) < r \cdot (r^2)^r (r|f(y)|)^{4r+\epsilon} < (r|f(y)|)^{8r}.$$

The computations in Lines 3 and 5 dominate the running time. Line 3 can be computed in $O(r^2 \cdot r^3 \log^{2+\epsilon}|h(x)|) = O(r^{7+\epsilon}\log^{2+\epsilon}(r|f(y)|))$ steps, as can Line 5.

Observe that if $s(x)$ is the minimal polynomial for θ over k, then

$$|s(x)| < r\binom{r}{r/2}||\alpha||^r < r2^r(r|f(y)|)^{8r} < (r|f(y)|)^{8r+\epsilon}.$$ □

We are now ready to compute fixed fields. Without loss of generality, suppose that $\{\sigma_1, \cdots, \sigma_l\}$ are the elements of H. Let $\Sigma a_j \theta_j$, with a_j in k, be an arbitrary element of L^H. For each σ_i in H, we have

$$\Sigma a_j \theta_j = \sigma_i(\Sigma a_j \theta_j) = \Sigma a_j \sigma_i(\theta_j).$$

Since the Galois group sends roots of $s(x)$ to roots of $s(x)$, we have $\sigma_i(\theta_j) = \theta_l$ for some l. Because the θ_j are linearly independent over k, the only way the above equation can be satisfied is if $a_i = a_l$. Each element σ_i in H gives rise to equalities amongst the a_i, which leads to a series of relationships amongst the θ_i. That is, if $a_i = a_l$, then in L^H any expression containing θ_i must have θ_l appearing with the same coefficient. Computing all such relationships gives us exactly the fixed field associated with H. We make this precise in the following algorithm:

ALGORITHM: COMPUTE FIXED FIELD
input: $\theta = h(a)$, an element in $k[x]/g(x)$ whose conjugates form a normal basis for $k[x]/g(x)$ over k, and $\{\sigma_1, \cdots, \sigma_l\} = H$, a subgroup of Galois group $G = \{\sigma_1, \cdots, \sigma_s\}$ of $k[x]/g(x)$ over k.
output: $\{\gamma_1, \cdots, \gamma_r\}$, where $(k[x]/s(x))^H = k(\gamma_1, \cdots, \gamma_r)$.

Line 1: BEGIN
Line 2:

$$a_{ij} = \begin{cases} 1 & \text{if } \sigma_i(h(a)) = \sigma_j(h(a)), \\ 0 & \text{otherwise;} \end{cases}$$

Line 3: $T \leftarrow$ transitive closure of (a_{ij});
Line 4: Let $C_1, \cdots, C_n$ be the connected components of the graph given by A:

$$\gamma_i = \Sigma_{j \in C_i} \sigma_j(h(a));$$

Line 5: END.

THEOREM 5.4. *Algorithm Compute Fixed Field computes the fixed field of a given subgroup H. It does so in $O(r^5 \log^2(r|f(y)|))$ steps.*

Proof. If $K = L^H$, then $K = k(\gamma_1, \cdots, \gamma_r)$, where the γ_j are sums of the θ_i. (Note that the γ_j partition the θ_i.) In Algorithm Compute Fixed Field, we set up an $n \times n$ matrix (a_{ij}), with 1's on the diagonal, 0's elsewhere. If σ_i in H takes θ_j to θ_k, then a_{jk} and a_{kj} are both assigned a 1. This is shorthand for saying that θ_j and θ_k appear with the same coefficient in K.

In Line 2 we set up the matrix. We compute the transitive closure in Line 3. Then in Line 3 we set up the γ_i, $i = 1, \cdots, n$, to be the sum of all the θ_i that have the same coefficient. We do this by computing the transitive closure of A. Clearly, $K = k(\gamma_1, \cdots, \gamma_r)$.

The computation is dominated by the calculations of Line 2 which take $O(r^2 \cdot r \log^2 |s(x)|) = O(r^5 \log^2(r|f(y)|))$ steps since $|s(x)| < (r|f(y)|)^{8r+\epsilon}$. □

At this point we observe that it is easy to calculate the minimal polynomial for a primitive lth root of unity. If $k = Q$, it will be the irreducible factor of $x^l - 1$ of degree $\varphi(l)$. If $l(t)$ is the minimal polynomial for ζ_l over k, then $|l(t)| < 2^t$. If $k \neq Q$, the minimal polynomial is a little more complicated to find, and there may be more than one choice for it (that is, there are nonconjugate primitive lth roots of unity in that case), but the bound on degree and coefficient size remains the same.

All that remains is to show how to transform a cyclic extension of degree l over k, where ζ_l, a primitive lth root of unity, is in k, into a radical extension. The technique was developed by Lagrange, two centuries ago.

DEFINITION. Let $L = k(\alpha)$ be a cyclic extension of degree l, and suppose that ζ_l is in k. Let σ be a generator of $G = Gal(L/k)$. For any element τ in L, define

$$(\zeta_l, \tau) = \tau + \zeta_l \sigma\tau + \cdots + \zeta_l^{l-1}\sigma^{l-1}\tau.$$

The element (ζ_l, τ) is called a *Lagrange resolvent.*

It is not hard to show that the σ's are linearly independent over k, thus there is a τ that makes the Lagrange resolvent nonzero. Such an element will generate L over k, and in particular will satisfy an irreducible polynomial over k of the form $x^l - b$. Observe that if $\theta_1, \cdots, \theta_l$ is a normal basis for L, then (ζ_l, θ) is nonzero, and this will suffice.

ALGORITHM: COMPUTE DENESTING.
input: $f(x)$, an irreducible polynomial of degree n over k, and n, the lcm of the indices of the reducible radical expressions for α.
output: A sequence of fields K_i, $i = 1, \cdots, s$, and an expression for α, where

1. the expression for α, a root of $f(x)$ in K_s, that is within one of minimal nesting depth over $k(\zeta_l)$,
2. and l =lcm (n, exponent G/DG, exponent $DG/D^2G, \cdots$, exponent $D^{s-1}G/D^sG$), and ζ_l is a primitive lth root of unity.

Line 1: BEGIN
Line 2: Compute L, the splitting field of $f(x)$ over k;
Line 3: Compute $G =$ Galois group of L over k;
Line 4: $D^0 \leftarrow G$;
Line 5: $i \leftarrow 0$
Line 6: WHILE $D^iG \neq \{e\}$ DO:
Line 7: BEGIN
Line 8: $i \leftarrow i + 1$;
Line 9: $D^i \leftarrow \langle \sigma\tau\sigma^{-1}\tau^{-1} \mid \sigma, \tau \in D^iG \rangle$
Line 10: END;
Line 11: $l \leftarrow$ lcm (n, exponent G/DG, exponent $DG/D^2G, \cdots$, exponent $D^{s-1}G/D^sG$);
Line 12: Find $l(x)$, minimal polynomial for ζ_l over k;
Line 13: $K_0 \leftarrow k(\zeta_l)$;
Line 14: FOR $i = 1$ TO s DO:
Line 15: BEGIN
Line 16: $L_i \leftarrow L^{D^iG}$;
Line 17: $K_i \leftarrow L_i(\zeta_l)$;
Line 18: Write $D^{i-1}G/D^iG$ as a product of cyclic groups, $J_{i1} \times \cdots \times J_{it_i}$;
Line 19: FOR $j = 1$ TO t_i DO:
Line 20: BEGIN
Line 21: $\tilde{J}_{ij} \leftarrow \{e\} \times \cdots \times \{e\} \times J_{ij} \times \{e\} \times \cdots \{e\} \subset D^{i-1}G/D^iG$;
Line 22: $L_{ij} \leftarrow L_i^{\tilde{J}_{ij}}$;
Line 23: $K_{ij} \leftarrow L_{ij}(\zeta_l)$;
Line 24: $l_{ij} \leftarrow [K_{ij} : K_{i-1}]$;
Line 25: Compute a normal basis for K_{ij} over K_{i-1}, $\theta_{ij1}, \cdots, \theta_{ijl_{ij}}$;
Line 26: Pick $\sigma_{(ij)}$, a generator of $\tilde{J}_{ij}$;
Line 27: $\beta_{ij} \leftarrow (\zeta_{l_{ij}}, \theta_{ij1})$;
Line 28: $K_{ij} \leftarrow K_{i-1}(\beta_{ij})$;
Line 29: END;
Line 30: $K_i \leftarrow K_{i-1}(\beta_{i1}, \cdots, \beta_{it_i})$;

Line 31: END;
Line 32: Express α in K_s;
Line 33: END.

THEOREM 5.5. *Let $f(x)$ of degree m be the minimal polynomial for α, a nested radical over k, and let L be the splitting field $f(x)$ over k, with Galois group G. Let D^iG represent the ith commutator subgroup of G, with $D^sG = \{e\}$, and let $l = lcm\,(n, exponent\,G/DG, \cdots, exponent\ D^{s-1}G/D^sG)$. On input $f(x)$, Algorithm Compute Denesting computes a denesting of α that is within depth one of optimal over $k(\zeta_l)$, where ζ_l is a primitive lth root of unity. It does so in $O(r^{12+\epsilon}\log^{4+\epsilon}(r|f(y)|))$ steps, where r is the order of the Galois group G.*

Proof. As always, we prove correctness, then analyze running time. We want a minimal depth denesting for α. By Theorem 4.7 there is a denesting of α over $k(\zeta_l)$ that is within depth one of minimal and which has each of its terms lying in $L(\zeta_l)$, where L is the splitting field of the minimal polynomial of α over k.

Thus what we must do is calculate L, the splitting field of $f(x)$ over k, G, its Galois group, the derived series, and l, the lcm of the indices of the reducible radicals in the original exression for α, and the exponents of the derived series. Then if $L_1 \subset L_2 \subset \cdots \subset L_s \supset L(\alpha)$ is a series of abelian extensions, so is $K_1 = L_1(\zeta_l) \subset L_2 = k_2(\zeta_l) \subset \cdots \subset K_s = L_s(\zeta_l)$ by Lemma 4.6. More to the point, by Lemma 4.6 K_i is a composite of radical extensions of K_{i-1} for each i. In particular, α will have an expression that is within one of minimal nesting depth in K_s.

In Line 2 we calculate the Galois group. It is straightforward to compute G, its derived series, the exponent l. This takes us to Line 12. Line 13 simply begins the process that will create the fields $K_i = L_i(\zeta_l)$ by setting $K_0 = k(\zeta_l)$.

In Line 16 we compute the fields $L_i = L^{D^iG}$, which correspond to the subgroups of the derived series. In Line 16 we create the corresponding tower of fields $K_i = L_i(\zeta_l)$. What we need to do next is compute the tower of fields K_i and express them as radical extensions.

We begin by computing the groups $D^iG/D^{i-1}G$ as a product of cyclic groups $J_{i1} \times \cdots \times J_{it_i}$. The we cycle through and compute first the groups $\tilde{J}_{ij} = \{e\} \times \{e\} \times J_{ij} \times \{e\} \times \cdots \{e\}$, and then the fixed field associated with each such $\tilde{J}_{ij}, L_{ij}$. By Lemma 4.6, the field calculated in Line 23, $K_{ij} = L_{ij}(\zeta_l)$ is a radical extension of L_{i-1}. In Line 25 we compute the normal basis $\theta_{i1}, \cdots, \theta_{il_j}$ for K_{ij} over K_{j-1}, and in Line 26 we pick an element from the associated Galois groups $\tilde{J}_{ij}$. The element $\beta_{ij} = \theta_{ij_1} + \zeta_{l_{ij}}\sigma_{ij}^{l_{ij}}\theta_{ij_1} + \cdots + \zeta_{l_{ij}}^{l_{ij}-1}\sigma_{ij}\theta_{i1}$ for some ordering of the θ's. Since the θ's are linearly independent, the element is perforce nonzero, hence by the theory of Lagrange resolvents, generates K_{ij} over K_{j-1}. In particular, β_{ij} satisfies an irreducible cyclic polynomial over K_{j-1}.

How long does the computation take? We claim no more than $O(r^{12+\epsilon}\log^{4+\epsilon}(r|f(y)|))$ steps, where $r = [L:k]$. Certainly the computations in Lines 1–11 are dominated by the time it takes to compute the Galois group; by Theorem 5.1, that can done in $O(r^{12+\epsilon}\log^{2+\epsilon}(r^2|f(y)|))$ steps. Lines 12 and 13 are a simple computation (even when $k \neq Q$). We run through the loop in Lines 14–29 at most $O(\log r)$ times. Its running time is dominated by Line 25, which takes $O(r^{8+\epsilon}\log^{2+\epsilon}|f(y)|)$ steps. The computation in Line 32 takes no time at all.

To simplify the computations, we view the fields K_i as simple extensions of K, and do the computation in terms of a primitive element, with minimal polynomial $g_{ij}(t)$ over k. However, since what we want is a basis for L with a minimal depth of nesting, we will store two different bases for K_{ij}; one with the β_{ij} that come from

the computations, and then $K_{ij} = k[t]/g_{ij}(t)$ written as an extension by a primitive element. We do not run into coefficient blowup in computing $g_{ij}(t)$ because each of the fields K_{ij} can be viewed as a subfield of $L(\zeta_l)$. (In the interests of making the algorithm as clear as possible, we have not included those fine points in the algorithm.) □

We have presented the running time analysis for the algorithm over Q. The only reason for that is simplicity. The only requirement for the base field is that one needs to be able to factor polynomials over k. Theorem A.4 shows how a factorization over k can be raised to a factorization over $k(\alpha)$. We can generalize Algorithm Compute Polynomial to compute a minimal polynomial for α over k. Of course, we can generalize Theorem 5.1, and Algorithms Compute Fixed Field and Compute Denesting.

Note also that the running time for Algorithm Compute Denesting can be simply stated as the time required to compute the splitting field of $f(x)$ over Q, which is in turn the time needed to factor $f(x)$ over its splitting field.

6. Conclusions. A number of open questions remain.

- Can the bound in Theorem 4.7 be made optimal, that is, can we improve it to $s \leq t$ without involving the form of the input (as per Corollary 4.8)? What is the trade-off between this and the roots of unity?
- Suppose α is a nested radical over a field k, and suppose there is a denesting expression for α involving roots of unity. When is there a way to transform that to an expression of the same depth which avoids using the roots of unity? (The expression

$$\sqrt{\sqrt{5} - 5/2} = \zeta_5 - 1/\zeta_5$$

makes it clear it that will not always be possible to do so.)

Our algorithm is not fast. Thus the other important issue in this problem is speed. We have several comments regarding this.

- Is there a way to perform these computations via a straightline program? The encoding in the straightline version would avoid the problem of coefficient blowup. The main difficulty seems to be in computing the Galois group. We do not see how to determine the group without determining both $f(x)$, the minimal polynomial of α over k, and the splitting field of $f(x)$ over k. If a way could be found around these problems, then the entire algorithm could be sped up. We do not think this will be easy. In particular, it is likely that any technique that works here will also work to determine the Galois group in the solvable case. No polynomial time algorithms are presently known, and this would be a surprising and exciting breakthrough.
- It is important to remember that the bounds given here are really *upper* bounds. The running time is exponential *if and only if* the splitting field is of exponential degree over k. If the degree of the splitting field is polynomially bounded, then so is the running time of the algorithm. Recall also that nearly all of the bounds follow from the factoring bound given in [11], and that bound is almost certainly not optimal. It is important to remember that what is theoretically slow may be practically fast, and vice versa. In particular, the exponential time polynomial factorization algorithm of [9] is more practical than any of the current polynomial time algorithms for polynomial factorization. Any improvements in the running time for factoring polynomials will lead to great improvements in our bounds.

- Our algorithm does not specifically examine the issue of determining if the expression equals zero. (We answer it, of course, in computing the minimal polynomial for α over k, but not efficiently.) Examining it leads to some interesting open questions on linear combinations of nested radicals. The case of nested square roots was treated by Borodin, Fagin, Hopcroft, and Tompa [3].

A. Appendix. As we stated earlier, we can view a nested radical α as a sequence of polynomials $\tilde{p}_i, \tilde{q}$ in $k[x_1, \cdots, x_{m-1}]$ such that

$$\alpha_1 = \sqrt[n_1]{\tilde{p}_1},\ \tilde{p}_1 \in k, \tag{4}$$

$$\alpha_2 = \sqrt[n_2]{\tilde{p}_2(\alpha_1)}, \tag{5}$$

$$\alpha_3 = \sqrt[n_3]{\tilde{p}_3(\alpha_1, \alpha_2)}, \tag{6}$$

$$\vdots \tag{7}$$

$$\alpha_m = \tilde{q}(\alpha_1, \cdots, \alpha_{m-1}) + \sqrt[n_m]{\tilde{p}_m(\alpha_1, \cdots, \alpha_{m-1})}, \tag{8}$$

with $\alpha = \alpha_m$. We show how to compute the minimal polynomial of $\alpha = \alpha_m$ over k. This polynomial is polynomial size in $n = \Pi n_i$, $||\alpha_i||$, $||\tilde{q}(x_1, \cdots, x_{m-1})||$, and $||\tilde{p}_i(x_1, \cdots, x_{i-1})||$. Whether or not the minimal polynomial is polynomial in the length of the expression for α is an open question.

We begin with some background. In particular, we define the size of a polynomial over Z, and over an algebraic number field, and quote some important results on polynomial factorization in these domains.

DEFINITION. Let $f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$ be a polynomial in $Z[x]$. The size of $f(x)$, denoted $|f(x)|$, is defined to be $(\Sigma a_i^2)^{1/2}$.

THEOREM A.1 (Mignotte [12]). *Let $f(t)$ be a primitive polynomial in $Z[t]$, and let $g(t)$, also primitive, be a factor of $f(t)$. Then $|g(t)| \leq 2^t |f(t)|$.*

DEFINITION. Let $h(x_1, \cdots, x_n) = h_{\alpha_1, \cdots, \nu_1} x_1^{\alpha_1} \cdots x_n^{\nu_1} + \cdots + h_{\alpha_t, \cdots, \nu_t} x_1^{\alpha_t} \cdots x_n^{\nu_t}$ be a polynomial over $O_k[x_1, \cdots, x_n]$. Then the size of $h(x_1, \cdots, x_n)$, which we will denote by $||h(x_1, \cdots, x_n)||$, is $(\Sigma h^2_{\alpha_j, \cdots, \nu_j})^{1/2}$.

THEOREM A.2 (Lenstra, Lenstra, and Lovász [11]). *Let $g(t) = a_n t^n + \cdots + a_0$ be a primitive polynomial of degree n over $Z[t]$. There is an algorithm to factor $g(t)$ into irreducible factors over $Z[t]$, which requires $O(n^{9+\epsilon} + n^{7+\epsilon} \log^{2+\epsilon} |g(t)|)$ steps.*

DEFINITION. Let β be an algebraic number. We define the *size* of β, written $||\beta||$, to be the maximum of the absolute values of the conjugates of β.

DEFINITION. Let $g(t)$ be a monic irreducible polynomial over Z, with roots $\alpha_1, \cdots, \alpha_m$. If $f(x) = \beta_n x^n + \cdots + \beta_0$, with $\beta_i = \Sigma_{j=0}^{m-1} b_{ij} \alpha^j$, then the *size* of $f(x)$, written $||f(x)||$, is defined to be the $\max_i (\Sigma_{j=0}^{m-1} b_{ij}^2)^{1/2}$.

THEOREM A.3 (Weinberger and Rothschild [16]). *Let $g(t)$ be a monic irreducible polynomial of degree m over Z, and let $K = Q[t]/g(t)$. Let β be a root of $f(x)$, a polynomial in $O_K[x]$. Then $||\beta|| \leq 1 + ||f(x)||$. Assume that $f(x)$ is monic, and that $h(x) = h_r x^r + \cdots h_0$ is a factor of $f(x)$ in $O_K[x]$. Then $||h(x)|| \leq m||f(x)||(m|g(t)|)^m$.*

THEOREM A.4 (Landau [6]). *Let $g(t)$ be a monic irreducible polynomial of degree m over Z, and let $K = Q[t]/g(t)$. Let $f(x)$ be a polynomial of degree n over O_K. Then there is an algorithm to factor $f(x)$ into irreducible factors over O_K. It runs in $O(m^{9+\epsilon} n^{7+\epsilon} \log^2 |g(t)| \log^{2+\epsilon}(||f(x)||(m|g(t)|)^n (mn)^n))$ steps.*

In order to calculate the minimal polynomial for α over k, we introduce two important concepts from number theory: the norm and the resultant. The norm

relates elements in extension fields to elements in the base field. Let $\beta = a_0 + a_1\gamma + \cdots + a_{m-1}\gamma^{m-1}$ be an element of the field $k(\gamma)$. Then

$$Norm_{k(\gamma)/k}(\beta) = N_{k(\gamma)/k}(\beta) = \Pi_i(a_0 + a_1\gamma_i + \cdots + a_{m-1}\gamma_i^{m-1}),$$

where the product is over the conjugates of γ. If σ is an element of the Galois group of the minimal polynomial of γ over k, then $\sigma(\gamma) = \gamma_j$ for some conjugate γ_j of γ. Then

$$\begin{aligned}\sigma_j(N_{k(\gamma)/k}(\beta)) &= \sigma_j(\Pi_i(a_0 + a_1\gamma_i + \cdots a_{m-1}\gamma_i^{m-1})) \\ &= \Pi_i\sigma_j(a_0 + a_1\gamma_i + \cdots + a_{m-1}\gamma_i^{m-1}) \\ &= \Pi_i(a_0 + a_1\gamma_i + \cdots + a_{m-1}\gamma_i^{m-1}) \\ &= N_{k(\gamma)/k}(\beta).\end{aligned}$$

Since σ_j just permutes the γ_i's, we conclude that $N(\beta)$ is in k. Further, the norm is multiplicative. We can extend the definition of norm to include polynomials, by thinking of $f(x)$ in $k(\gamma)[x]$ as a polynomial in two variables: x and γ, and we denote it by $f_\gamma(x)$. Then

$$N_{k(\gamma)/k}(f(x)) = \Pi_i f_{\gamma_i}(x).$$

If $f(x)$ is in $k(\gamma)[x]$, then the norm of $f(x)$, $N_{k(\gamma)/k}(f(x))$, is in $k[x]$. We will need the following important property of norms of polynomials.

LEMMA A.5 (Weyl [17], Trager [15]). *If $f(x)$ is an irreducible polynomial over $k(\gamma)$, then $Norm_{k(\alpha)/k}(f(x))$ is the power of an irreducible polynomial over k.*

How do we calculate the norm of a polynomial? The coefficients of the norm are all symmetric functions in γ_i. A simple algorithm for computing norms using determinants has been known since the nineteenth century; this is the *resultant.* Let $g(t) = g_m t^m + g_{m-1}t^{m-1} + \cdots + g_0$, and $f(t) = f_n t^n + \cdots + t_0$. We define $Res_t(g(t), f(t)) =$

$$\begin{vmatrix} f_n & 0 & 0 & \cdots & 0 & g_m & 0 & 0 & \cdots & 0 \\ f_{n-1} & f_n & 0 & \cdots & 0 & g_{m-1} & g_m & 0 & \cdots & 0 \\ f_{n-2} & f_{n-1} & f_n & \cdots & 0 & g_{m-2} & g_{m-1} & g_m & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ f_0 & f_1 & f_2 & \cdots & f_{m-1} & g_{m-n} & g_{m-n+1} & g_{m-n+2} & \cdots & g_m \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & \cdots & 0 & f_0 & 0 & 0 & \cdots & 0 & g_0 \end{vmatrix}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{m}\underbrace{\qquad\qquad\qquad\qquad}_{n}$$

The coefficients $g_i, f_j = 0$ whenever $i, j < 0$, respectively. It can be shown that $Res_x(g(x), f(x)) = g_m^n \Pi f(\gamma_i)$. Thus $N(f(x)) = \Pi f_{\gamma_i}(x) = (Res_t(g(t), f(x,t)))/g_m^n$, where $f(x,t)$ is $f(x)$ with t's substituted in for γ's, and $g(t)$ is the minimal polynomial for γ over k.

The nesting expressions for α in (4)–(8) is general, but cumbersome. Instead of multivariate polynomials $\tilde{q}, \tilde{p}_i$, we would prefer to have univariate polynomials q, p_i. We can do this by using Lemma A.6.

LEMMA A.6 (Trager [15]). *Suppose* $[k(\alpha,\beta) : k(\alpha)] = n$, *and* $[k(\alpha) : k] = m$. *Then there is a* $c < (mn)^2$ *such that* $k(\alpha,\beta) = k(\alpha+c\beta)$. *If* $h(x)$ *is the minimal polynomial for* β *over* $k(\alpha)$, *then* $k(\alpha,\beta) \simeq k[x]/H(x)$ *whenever* $H(x) = Norm_{k(\alpha)/k}(h(x - c\alpha))$ *is square-free.*

LEMMA A.7 (Landau [6]). *Let* k, α, β, c, m, *and* n *be as above. If* $g(x)$ *is the minimal polynomial for* α *over* Q, *then* $H(x)$, *the minimal polynomial for* $\alpha + c\beta$ *over* Q, *has coefficients no larger than* $(c|g(x)|||h(x)||)^{m+n}$.

Thus for each α_i there is an associated γ_i, with $\gamma_1 = \alpha_1$, and $\gamma_i = \alpha_i + c_i\gamma_{i-1}$. Then $k(\alpha_1, \cdots, \alpha_i) = k(\gamma_i)$. The multivariate polynomials in (4)–(8), $\tilde{q}$, $\tilde{p}_i$, can be replaced by univariate polynomials q, p_i, and a nesting expression can be more simply written as:

$$\begin{aligned}
\alpha_1 &= \sqrt[n_1]{p_1},\ p_1 \in k,\\
\alpha_2 &= \sqrt[n_2]{p_2(\gamma_1)}, \gamma_1 = \alpha_1,\\
\alpha_3 &= \sqrt[n_3]{p_3(\gamma_2)}, \gamma_2 = \alpha_2 + c_2\gamma_1,\\
&\ \vdots\\
\alpha_m &= q(\gamma_{m-1}) + \sqrt[n_m]{p_m(\gamma_{m-1})}, \gamma_{m-1} = \alpha_{m-1} + c_{m-1}\gamma_{m-2},
\end{aligned}$$

with the $c_i < (\Pi_{j<i} n_j)^2$. Lemma A.6 gives a method for computing the minimal polynomial of γ_i over k. We will show how to compute these polynomials in Algorithm Compute Polynomial, where we also compute $f(x)$, the minimal polynomial for α over k. We will first sketch the idea behind the algorithm.

It is easy to see what the minimal polynomial for α_1 over k is. Suppose $\tilde{p}_i(x)$ is a dth power of an element in k, say, p_1, and let d be the largest integer dividing n_1 for which this statement is true. If we let $a_1 = n_1/d$, then $\sqrt[n_1]{\tilde{p}_1} = \sqrt[a_1]{p_1}$, and α_1 satisfies $f_1(x) = x^{a_1} - p_1$. If this polynomial is irreducible, we are done. We could factor to check irreduciblity, but a more efficient test would be to use the criterion of Theorem 2.9. If the conditions of Theorem 2.9 are not satisfied (i.e., $4 \mid a_1$, ζ_{a_1} is not in k, $p_1 = -4j^4$ for some j in k), then $x^{a_1} - p_1$ is reducible. In this situation, we do factor $x^{a_1} - p_1 = \Pi j_i(x)$, and set $f_1(x)$ to be the $j_i(x)$ for which $j_i(p_1) = 0$. Of course, $g_1(t)$, the minimal polynomial for $\sqrt[a_1]{p_1(q)}$ over k, is equal to $f_1(t)$.

To compute the minimal polynomial for α_2 over k, we do nearly the same computation. We find d, the largest integer dividing n_2 such that $\tilde{p}_2(\alpha_1)$ is a dth power of an element in $k(\gamma_1)$, say, $p_2(q)$. Again, $a_2 = n_2/d$, $\sqrt[n_2]{\tilde{p}_2(\alpha_1)} = \sqrt[a_2]{p_2(q)}$, and α_2 satisfies $x^{a_2} - p_2(\gamma_1)$. We check irreducibility as before, and set $f_2(x)$ equal to the minimal polynomial of α_2 over $k(\gamma_1)$. By Lemma A.5, the polynomial

$$s_2(x) = N_{k(\gamma_1)/k}(f_2(x)) = Res_t(t^{a_1} - p_1(q), f_2(x,t))$$

is a power of an irreducible polynomial over k which α_2 satisfies. If $h_2(x)$ is that polynomial, then $h_2(x) = s_2(x)/\gcd(s_2(x), s_2'(x))$.

Next we compute the minimal polynomial of γ_2 over k. (Note that that is really equivalent to computing a primitive element of $k(\gamma_1, \alpha_2)$ over k.) By Lemma A.6, it suffices to find a c_2 such that $g_2(t) = N_{k(\gamma_1)/k}(f_2(t - c_2\gamma_1))$ is square-free, and there is a $c < (a_1a_2)^2$ that will work. We calculate the norm by

$$N_{k(\gamma_1)/k}(h_2(t - c_2\gamma_1)) = Res_y(g_1(y), h_2(t - c_2y, y)),$$

thinking of $h_2(t)$ in $k(\gamma_1)$ as a polynomial in t and y in $k[t,y]/g_1(y)$.

At this point, the pattern begins to emerge. We are first calculating the minimal polynomial of α_i over $k(\gamma_{i-1})$, either $x^{a_i} - p_i(\gamma_{i-1})$, or a divisor of it. Call it $f_i(x)$. Then $s_i(x) = Res_t(g_{i-1}(t), f_i(x,t))$ is the power of an irreducible polynomial over k, $h_i(x) = s_i(x)/\gcd(s_i(x), s_i'(x))$. We then compute a primitive element of $k(\gamma_i) = k(\gamma_{i-1}, \alpha_i)$ over k. We do that by finding a c_i such that $g_i(t) = N_{k(\gamma_{i-1})/k}(f_i(t - c_i\gamma_{i-1}))$ is square-free. We are then ready to repeat the process.

ALGORITHM: COMPUTE POLYNOMIAL.
input: $(\tilde{q}_i(x_1, \cdots, x_{i-1}), n_i, \tilde{p}_i(x_1, \cdots, x_{i-1})), i = 1, \cdots m$, where $\alpha_1 = \sqrt[n_1]{\tilde{p}_1}, \alpha_2 = \sqrt[n_2]{\tilde{p}_2(\alpha_1, \alpha_2)}, \cdots,$
$\alpha = \alpha_m = \tilde{q}(x_1, \cdots x_{m-1}) + \sqrt[n_m]{\tilde{p}_m(\alpha_1, \cdots, \alpha_{m-1})}$.
output: $f(x)$, the minimal polynomial of α over k, and $g_i(t)$, where $k(\alpha_1, \cdots, \alpha_i) \simeq k[t]/g_i(t)$ for $i = 1, \cdots, m$, and n =lcm of the indices of the reducible radicals $\hat{p}_i$.

Line 1: BEGIN
Line 2: Compute minimal polynomial for $\tilde{p}_1$, and call it $h_1(x), f_1(x)$;
Line 3: $g_1(t) \leftarrow f_1(t)$;
Line 4: $n \leftarrow 1$;
Line 5: FOR $i = 2$ TO $m - 1$ DO:
Line 6: BEGIN
Line 7: Express $\alpha_1, \cdots, \alpha_{i-1}$ as elements in $k[t]/g_{i-1}(t)$ and Rewrite $\tilde{p}_i(\alpha_1, \cdots, \alpha_{i-1})$ as $p_i(t)$;
Line 8: Find $f_i(x)$, minimal polynomial for $p_i(t)$ over $k[t]/g_{i-1}(t)$;
Line 9: Write "α_i satisfies $f_i(t)$ over $k[t]/g_{i-1}(t)$"; (This is either $x^{n_i} - p_i(t)$, or a divisor of it; we will pick a divisor of minimal degree.)
Line 10: If $\hat{p}_i$ is a reducible radical, $n \leftarrow \text{lcm}(n_i, n)$;
Line 11: $s_i(x) \leftarrow Res_t(g_{i-1}(t), f_i(x))$;
Line 12: $h_i(x) \leftarrow s_i(x)/\gcd(s_i(x), s_i'(x))$;
Line 13: Find c_i such that $g_i(t) \leftarrow Res_y(g_{i-1}(y), f_i(t - c_i y, y))$ is square-free;
Line 14: END;
Line 15: Rewrite $\tilde{p}_m(\alpha_1, \cdots, \alpha_{m-1})$ as $p_m(t)$, and rewrite $\tilde{q}(\alpha_1, \cdots, \alpha_{m-1})$ as $q(t)$;
Line 16: Find $f_m(x)$, minimal polynomial for $q(t) + \sqrt[n_m]{p_m(t)}$ over $k[t]/g_{m-1}(t)$; (This is either $(x - \tilde{q}(t))^{n_m} - \tilde{p}_m(t)$, or a divisor of it.)
Line 17: $s_m(x) \leftarrow Res_t(g_{m-1}(t), f_m(x))$;
Line 18: $f(x) \leftarrow s_m(x)/\gcd(s_m(x), s_m'(x))$;
Line 19: Find c_m such that $g_m \leftarrow Res_t(g_{m-1}(y), f_m(t - c_m y, y))$ is square-free;
Line 20: END.

THEOREM A.8. *On input the sequence* $\alpha_1 = \sqrt[n_1]{\tilde{p}_1}, \cdots, \alpha_m = \tilde{q}(\alpha_1, \cdots, \alpha_{m-1}) + \sqrt[n_m]{\tilde{p}_m(\alpha_1, \cdots, \alpha_{m-1})}$, *where the* $\tilde{p}_i, \tilde{q}$ *are polynomials in* $O_k[x_1, \cdots, x_m]$ *for* $i = 2, \cdots, m$, *and* $\tilde{p}_1$ *is in* O_k, *the Algorithm Compute Polynomial computes* $f(x)$, *the minimal polynomial for* $\alpha = \alpha_m$ *over* k, *and the polynomials* $g_i(t)$, $i = 1, \cdots, m$, *where* $k(\alpha_1, \cdots, \alpha_i) \simeq k[t]/g_i(t)$. *It does so in* $O(n^{21} \log^{4+\epsilon}(\mathcal{AP}))$ *steps, where* $\mathcal{A} = \max_i ||\alpha_i||$, *and* $\mathcal{P} = \max(||\tilde{p}_i(x_1, \cdots, x_{i-1})||, ||\tilde{q}(x_1, \cdots, x_{m-1})||)$.

Proof. As usual, we begin with a proof of correctness. We will prove that $g_i(t)$ in $O_k[t]$ is an irreducible polynomial generating $k(\alpha_1, \cdots, \alpha_i)$, i.e., that $k(\alpha_1, \cdots, \alpha_i) \simeq k[t]/g_i(t)$, that $f_i(x)$ is the minimal polynomial of α_i over $k[t]/g_{i-1}(t)$, and that $h_i(t)$ is the minimal polynomial for α_i over k.

That $f_1(x)$, $h_1(x)$, and $g_1(t)$ calculated in Lines 2 and 3 satisfy these conditions is clear. We prove by induction that the $f_i(x)$, $h_i(x)$, and $g_i(t)$ calculated in Lines 4–14 do so also. It is clear that α_i satisfies $x^{a_i} - p_i(t)$. By using the criterion of

Theorem 2.9, and factoring if necessary, we find $f_i(x)$, the minimal polynomial of α_i over $k[t]/g_{i-1}(t)$.

Since $g_{i-1}(t)$ generates $k(\alpha_1, \cdots, \alpha_{i-1})$ over k, we have that $Res_t(g_{i-1}(t), f_i(x)) = N_{k(\alpha_1,\cdots,\alpha_{i-1})/k}(f_i(x))$ is a polynomial over k which α_i satisfies. Furthermore, by Lemma A.5, $s_i(x) = Res_t(g_{i-1}(t), f_i(x))$ is the power of an irreducible polynomial over k. Thus $h_i(x)$ is the minimal polynomial of α_i over k. Finally, by Lemma A.6, the $g_i(t)$ calculated in Line 13 is the minimal polynomial of $\alpha_i + c_i\gamma_{i-1}$, where γ_{i-1} is a root of $g_{i-1}(t)$, and thus generates $k(\alpha_1, \cdots, \alpha_i)$, that is, $k(\alpha_1, \cdots, \alpha_i) \simeq k[t]/g_i(t)$.

How long does the computation take? In order to discuss the time issue, we need to know how big the coefficients of $f_i(x)$, $s_i(x)$, $h_i(x)$, and $g_i(t)$ can be. We analyze that first. Our bounds are not tight, but are chosen for the relative simplicity of the analysis.

Let $n =$ degree $(f(x))$, and let $\mathcal{A} = \max_i ||\alpha_i||$. Observe that $h_i(x)$ has root α_i over k, and that thus each coefficient of $h_i(x)$ is less than $2^n||\alpha_i||^n$ by the binomial theorem. Therefore $|h_i(x)| < (n(2^n||\alpha_i||^n)^2)^{1/2} < (2\mathcal{A})^{n+\epsilon}$. Then $\gamma_i = c_1 \cdots c_i\alpha_1 + c_1 \cdots c_{i-1}\alpha_2 + \cdots + \alpha_{i-1}$, a root of $g_i(t)$, is less than $n^{2n}\mathcal{A}$, and the coefficients of $g_i(t)$ are less than $2^n(n^{2n}\mathcal{A})^n$. Thus $|g_i(t)| < (n(2^n(n^{2n}\mathcal{A})^n)^2)^{1/2} < (n\mathcal{A})^{4n^2+\epsilon}$.

It is easy to analyze the size of $s_i(x)$. Observe that all the irreducible factors of $s_i(x)$ appear in $h_i(x)$. Thus $|s_i(x)| < 2^n|h_i(x)|^n < (4\mathcal{A})^{n^2+\epsilon}$.

Finally we are ready to tackle $f_i(x)$. Rewriting $\tilde{p}_i(\alpha_1, \cdots, \alpha_{i-1})$ as $p_i(t)$ will increase the coefficient size only slightly. This is because all we are doing is a matrix computation. Let $N_i = \Pi_{j\leq i} n_j$. The coefficients of $f_i(x)$ are bounded by $(|g_i(t)|||\tilde{p}_i(x_1, \cdots, x_{i-1})||)^{N_j} N_j! < n!\mathcal{A}^{4n^2+\epsilon}||\tilde{p}_i||^n$.

The running time of the algorithm is dominated by the factorizations of Lines 2, 8, 11, 13, and 16. The factorizations of Lines 8 and 16 are over algebraic extensions of k, and are thus most expensive. They take at most

$$O(n^{9+\epsilon}n^{7+\epsilon}\log^2(n\mathcal{A}^{4n+\epsilon})\log^{2+\epsilon}(n!\mathcal{A}^{4n^2+\epsilon}||\tilde{\mathcal{P}}||^n(n(n\mathcal{A})^{4n+\epsilon})^n(n^2)^n))$$
$$< O(n^{20+\epsilon}\log^{4+\epsilon}(\mathcal{A}\mathcal{P}))$$

steps, where $\mathcal{P} = \max(||\tilde{p}_i(x_1, \cdots, x_{i-1})||, ||\tilde{q}(x_1, \cdots, x_{m-1})||)$. The loop is repeated at most m times, hence the bound in the theorem. □

Acknowledgments. Warm thanks go to Tsuneo Tamagawa, for help with Theorem 4.7, and for many insights into Galois theory. Many thanks also to John Cremona, whose thorough reading and many observations greatly improved this paper, and to an anonymous referee whose careful reading of an earlier version caught an error.

REFERENCES

[1] E. ARTIN, *Galois Theory*, University of Notre Dame Press, South Bend, IN, 1942.

[2] L. BABAI, E. LUKS, AND Á. SERESS, *Fast management of permutation groups*, in Proc. 29th Annual Symposium on Foundations of Computer Science, 1988, pp. 272–282.

[3] A. BORODIN, R. FAGIN, J. HOPCROFT, AND M. TOMPA, *Decreasing the nesting depth of expressions involving square roots*, J. Symbol. Comput., 1 (1985), pp. 169–188.

[4] B. CAVINESS AND R. FATEMAN, *Simplification of radical expressions*, in Proc. 1976 ACM Symposium on Symbolic and Algebraic Computation, ACM, New York, 1976.

[5] D. KNUTH, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison–Wesley, Reading, MA, 1969.

[6] S. LANDAU, *Factoring polynomials over algebraic number fields*, SIAM J. Comput., 14 (1985), pp. 184–195.

[7] S. LANDAU AND G. MILLER, *Solvability by radicals is in polynomial time*, J. Comput. System Sci., 30 (1985), pp. 179–208.

[8] S. LANG, *Algebra,* Addison–Wesley, Reading, MA, 1971.

[9] A. K. LENSTRA, *Lattices and Factorization of Polynomials*, in Proc. Eurocam 1982, Lecture Notes in Computer Science 144, Springer-Verlag, Berlin, New York, pp. 32–39.

[10] ———, *Factoring polynomials over algebraic number fields*, in Proc. Eurocal 1983, Lecture Notes in Computer Science, Springer-Verlag, Berlin, New York, pp. 458–465.

[11] A. K. LENSTRA, H. W. LENSTRA, JR., AND L. LOVÁSZ, *Factoring Polynomials with Rational Coefficients*, Math. Ann., 261 (1982), pp. 513–534.

[12] M. MIGNOTTE, *Some inequalities about univariate polynomials*, in Proc. ACM Symposium on Algebraic and Symbolic Computation, 1981, pp. 195-199.

[13] J. ROTMAN, *The Theory of Groups*, Allyn and Bacon, Boston, MA, 1973.

[14] P. SAMUEL, *Algebraic Theory of Numbers*, Hermann, Paris, 1972.

[15] B. TRAGER, *Algebraic factoring and rational function integration*, in Proc. ACM Symposium on Symbolic and Algebraic Computations, 1976, pp. 219–226.

[16] P. WEINBERGER AND L. ROTHSCHILD, *Factoring Polynomials over Algebraic Number Fields*, ACM Trans. Math. Software, 2 (1976), pp. 335–350.

[17] H. WEYL, *Algebraic Theory of Numbers*, Princeton University Press, Princeton, NJ, 1940.

[18] R. ZIPPEL, *Simplification of expressions involving radicals*, J. Symbol. Comput., 1 (1985), pp. 189–210.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 111–139, February 1992
© 1992 Society for Industrial and Applied Mathematics
010

GOSSIPING IN MINIMAL TIME*

DAVID W. KRUMME†, GEORGE CYBENKO‡, AND K. N. VENKATARAMAN§

Abstract. The gossip problem involves communicating a unique item from each node in a graph to every other node. This paper studies the minimum time required to do this under the weakest model of parallel communication, which allows each node to participate in just one communication at a time as either sender or receiver. A number of topologies are studied, including the complete graph, grids, hypercubes, and rings. Definitive new optimal time algorithms are derived for complete graphs, rings, regular grids, and toroidal grids that significantly extend existing results. In particular, an open problem about minimum time gossiping in complete graphs is settled. Specifically, for a graph with N nodes, at least $\log_\rho N$ communication steps, where the logarithm is in the base of the golden ratio ρ, are required by any algorithm under the weakest model of communication. This bound, which is approximately $1.44 \log_2 N$, can be realized for some networks and so the result is optimal.

Key words. gossiping, broadcasting

AMS(MOS) subject classifications. 68Q20, 68R10

1. Introduction.

Gossiping generally refers to the process of distributed information dissemination and can easily be described in graph-theoretic terms. Each node in a graph initially contains a unique piece of information to be communicated to all other nodes. At each time step, a node can only communicate with those nodes that share an edge with it. Information can be combined between communications. Variants of the gossip problem involve the minimal total number of communications and the minimal total time required. Different models of communication have been proposed. Known results about gossiping are summarized in a 1988 survey paper by Hedetniemi, Hedetniemi, and Liestman [16].

In this paper, we study gossiping in minimum time under the weakest model of communication and derive a number of optimal results for various graph families. Our motivation for studying this problem is threefold and stems from the identification between multiprocessor interconnection schemes and graphs. First, gossiping challenges the data throughput capabilities of any interconnection graph while at the same time it is a model for a number of parallel communications problems. Second, minimum time solutions for gossiping provide lower bounds for the communication complexity of algorithms for a large class of problems. Finally, the efficiency of an optimal time algorithm for gossiping on a particular topology is a useful measure of that graph's parallel communication capability when viewed as an interconnection scheme. As such, it introduces what we believe to be a valuable metric in evaluating multiprocessor interconnection networks.

1.1. The gossip problem.

Our interest in the gossip problem stems from its relationship to communications problems in distributed memory multiprocessor systems. Distributed memory multiprocessors share data by passing messages along dedicated channels that connect pairs of processors. A prototypical example is the commercially successful binary hypercube architecture [15], [25]. These architectures

* Received by the editors March 16, 1988; accepted for publication (in revised form) February 20, 1991. This work was partially supported by Office of Naval Research contract N00014-87-K-0182 and National Science Foundation grant DCR-8619103.

† Department of Computer Science, Tufts University, Medford, Massachusetts 02155.

‡ Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois 61801.

§ Motorola Cambridge Research Ctr., 1 Kendall Sq. Bldg. 200, Cambridge Massachusetts 02139.

are idealized as graphs where nodes are processors with local memories and edges are direct communication channels. Any pair of processors can communicate but if they are not neighbors then messages have to be routed through the network and the time required is, to the first order, proportional to the length of the route used. Although our original motivation for studying gossiping came from working with multiprocessor systems, we use the language of graphs to simplify and generalize our discussions and results.

The gossip problem is easy to formalize. Each node in a graph has a *token*, or unit of data, that needs to be communicated to all other nodes in the graph. Tokens can be combined so that all communications involve constant time. The time needed for combining is irrelevant and treated as zero. A formal definition is simply stated:

Initialization: Let $G = (V, E)$ be a graph (interconnection network). With each node v, associate an initial singleton set $S(v)$ (the initial data). These initial singleton sets are disjoint.

Allowable Steps: Each node can send its set to a neighbor or neighbors and/or receive a set from a neighbor or neighbors depending on the model of communication used. After receiving any sets, nodes take the union of their existing sets with all sets received at that step, thus forming new sets for the next step.

Final State: All nodes must have the same sets locally, containing all elements in the initial singleton sets.

The only unspecified ingredient in the gossip problem as described above is the model of what a feasible communication is. There are two independent parameters that we believe can model most distributed memory systems realistically and we restrict discussion to them. They are: (a) the degree of a feasible communication step and (b) the duplex mode of communication channels. The degree of a feasible communication is the maximal number of simultaneous communication activities allowed at each node. We assume that this number is either one or the maximal degree possible (although intermediate situations are surely conceivable) and we refer to these two models as pairwise and simultaneous, respectively. The duplex mode is either full or half. This refers to whether simultaneous reading/writing can take place between a pair of connected nodes (processors).

We will use some abbreviated notation when referring to these models of communication. Specifically, F1 and H1 refer to the full-duplex and half-duplex pairwise models, respectively—the suffix of 1 indicates that a node can only communicate with one other node at a time. Similarly, F$*$ and H$*$ denote simultaneous communications models with full-duplex and half-duplex modes so that communication with any number of neighboring nodes is allowed. This leaves open the possibility of studying models like H2, H3, F2, F3, and so on with the obvious meaning that Hn allows n channels to be used simultaneously (see [14] for a discussion of a specific multiprocessor design allowing this). The gossip problem for the complete graph under the Fn model has been studied by Schmitt [24] and, under a restricted version of the Hn model, by Entringer and Slater [8].

A solution to this problem is just a sequence of feasible communication steps (an algorithm). Each communication between neighboring pairs of nodes takes one time unit. Our measure of the complexity of an algorithm for solving this problem is the number of time units required to complete the algorithm. When we discuss optimal algorithms we typically mean optimal to within an additive constant number of steps. However, in some instances we are actually interested in tight optimality, since we are

interested both in asymptotic complexity and in specific cases representative of real systems.

We study this problem on basic topologies that include linear arrays, regular and toroidal grids, rings, hypercubes, and complete graphs. Our results are primarily for the H1 model of communication, although we do mention results for other models as well. A recent paper of Bagchi, Hakimi, Mitchem, and Schmeichel [4] contains conjectures and results for grids, hypercubes, and rings under the H1 model, which we describe in §§6, 7, and 9, respectively.

One result derived in the paper settles an open problem in the field [8]. (Initial announcement of it appeared in [28]. Subsequently, Even and Monien [9] discovered the same result.) Assume the H1 model of communication so that in one time step, a node can only be engaged in one communication activity (either sending or receiving with one of its neighbors). Then a lower bound for solving the gossip problem for any graph consisting of N nodes is $\log_\rho N$ where the logarithm is in the base of the golden ratio $\rho = (1+\sqrt{5})/2$ so $\log_\rho N \approx 1.44 \log_2 N = 1.44 \lg N$ and this bound is optimal.[1] To contrast this with some well-known algorithms for the hypercube, note that a common solution on the hypercube uses $2 \lg N$, which is a constant factor of about $\frac{1}{3}$ worse than the optimal solution for an optimally connected graph. A companion paper to this one shows that $2 \lg N$, is not optimal for hypercubes in general; optimal algorithms for hypercubes in this model of communication are not known.

1.2. Background. There are a number of common situations in multiprocessing where gossiping occurs. One application is that of global processor synchronization, or a barrier type of construction [3]. All processors are to suspend execution at a certain breakpoint, or barrier, until such time as all other processors have reached the appropriate breakpoints in their executing programs.

The gossip problem is also an abstraction of a large class of distributed computation problems. Suppose that a parallel computation requires both input and output data to be distributed across the network. If the outputs require all inputs, then whatever algorithm is used, there is an implicit solution to the gossip problem in the communication pattern used by the algorithm. Optimal solutions to the gossip problem therefore provide lower bounds on the communication complexity of any algorithm for performing such a computation. Based on this observation, it should be evident that the gossip problem thus provides communications lower bounds for problems such as linear system solving, Discrete Fourier Transform evaluation, and sorting. For example, consider the problem of inverting a square $n \times n$ matrix where the entries of the matrix are distributed over n^2 processors. If the inverse matrix is to have elements distributed likewise, then our results show that under the H1 model described above, any algorithm running on any interconnection network will require at least $1.44 \lg n^2 = 2 \log_\rho n$ communications steps. This should be compared with algorithms requiring $O((\log n)^2)$ arithmetic operations for matrix inversion [7], [21].

The study of algorithms for the gossip problem on standard topologies naturally leads to asking whether there is an effective procedure for finding optimal algorithms for any interconnection network. We show that for the H1 and F1 models of communication this problem is NP-complete by a reduction of the broadcast problem of [11] to the gossip problem studied here. This is discussed in §4.

[1] Throughout the remainder of this paper we will use $\log_r n$ to denote the logarithm of n in the base r, $\rho = (1+\sqrt{5})/2$ to denote the golden ratio, and $\lg n = \log_2 n$.

1.3. Related research. The communication complexity of parallel algorithms is an area receiving more attention from the research community. The earliest work that we know of in this direction is due to Gentleman [12], who studied the communication complexity of matrix computations on grid interconnection networks like that in the ILLIAC IV. Gentleman used what we call the F$*$ model of communication and showed, for example, that at least $0.70n$ communication steps were needed to invert an $n \times n$ matrix stored in an $n \times n$ grid.

A variety of communications problems have recently been studied by researchers interested in optimal routing for data movement problems or specialized communications. Stout and Wager [27] study several communication problems for the binary hypercube under the F$*$ model. Saad and Schultz [23] compared the performance of algorithms for various communications problems on a variety of standard multiprocessor architectures. The model of communication used for distributed memory architectures is equivalent to the H$*$ model of this paper. Other recent works on communication complexity and optimal routing can be found in [2], [5], [22].

Our results on optimal lower bounds for complete interconnection graphs are the same as lower bounds for semioblivious PRAM machines computing functions with critical input [6]. Specifically, semioblivious PRAMs require at least $0.5 \log_\rho n \approx 0.72 \lg n$ PRAM-steps to compute a function with critical input. PRAMs allow simultaneous reading of memory locations but require serialization of writing, but this does not match any natural model of interprocessor communication in distributed memory systems.

1.4. Organization. This paper is organized as follows. Section 2 introduces and discusses our models of communication in more detail. Section 3 surveys results under three of the communication models (all but H1). The rest of the paper deals with the H1 model. Section 4 shows that the derivation of optimal time algorithms for gossiping on an arbitrary graph using the H1 or the F1 model is NP-complete. Section 5 deals with the important case of the complete graph. This case is fundamental because it provides a lower bound on the time required for gossiping on any graph and as mentioned above it provides lower bounds on the communication required to solve a large class of distributed computation problems. All of these topics are presented in §5. Sections 6–10 deal with specific interconnection topologies—linear array, rectangular grids, toroidal grids, hypercubes, and rings. Section 11 is a discussion.

2. Realistic models of communication cost. In a distributed memory multiprocessor, a communication transaction is an interplay of both hardware and software activity. As already noted, the H1 model of communication makes the weakest assumptions about both hardware and software capabilities. Detailed discussions of the relevance of the various models in real multiprocessor systems, such as hypercubes, can be found in the longer technical reports [19], [20]. We can summarize those observations by stating that the H$*$ and F$*$ models tend to reflect hardware characteristics, while the H1 and F1 models reflect software characteristics. A real system will reflect some combination, possibly a complex one, with perhaps either software or hardware effects dominating.

In the gossip problem it is assumed that tokens may be freely combined and transmitted as cheaply as one, or in other words, that the time required to transmit a message is a fixed constant independent of the length of the message. Cases where such assumptions would be valid include the following: when the messages are small, for example, if synchronization or control information is being passed, so that the dependence of transmission time on message size is negligible; when it is possible

to coalesce the message contents, such as when a global sum or maximum is being computed; where the transmission hardware or software uses very large packet sizes as on the first-generation Intel iPSC; and where local per-message overhead is high enough that actual message transmission time is negligible. Furthermore, results obtained under this assumption provide a lower bound for the case where transmission time does depend on message size.

We also assume that communication occurs as if globally synchronized, i.e., at each discrete time step, one set of communications across links in the network occurs. It suffices to note that the algorithms we describe can be viewed as self-synchronizing since the algorithms are deterministic and so each processing node can store the sequence of communications required and initiate a transmission only when that transmission's precedents have been completed.

Of the four models, we consider H1 to be the most important. It is the weakest—in fact, it represents the minimal communication capability that a loosely coupled multiprocessor could possibly have. Thus upper bounds developed under the H1 model apply to all models, while lower bounds under H1 serve as upper bounds for potential lower bounds under other models. It characterizes one real machine fairly well (the first generation NCUBE), especially for small messages. Algorithms developed under the H1 model will generally impose the least load on a system since they represent the smallest use of communication resources.

3. Full-duplex and simultaneous models. Analysis of the problem under the F$*$, H$*$, and F1 models is easy for most of the graphs we deal with. In this section we treat these three models in turn, leaving the H1 model for the rest of the paper. Before proceeding, we observe that the number of steps required under the H1 model is no more than twice what is required under the F1 model and no more than a factor of v times what is required under the H$*$ model, where v is the maximum degree of any node. Similarly, the complexity under the F1 model is no more than a factor of v times the complexity under the F$*$ model, and under the H$*$ model it is no more than twice that under the F$*$ model. Although this indicates a potential range from D (the diameter) under F$*$ to $2vD$ under H1, we shall find that solutions in close to D steps are almost always obtained under all four models.

3.1. Full-duplex simultaneous communication (F$*$). Under the F$*$ model, the gossip problem on *any* graph has the graph's diameter as a tight bound: each node can simply send its tokens to all its neighbors at each step, and the time needed to gossip will simply be the diameter.

3.2. Full-duplex pairwise communication (F1). For the complete graph with an even number N of nodes (and for the hypercube of N nodes), the time to gossip under the F1 model is the same as the time required to broadcast from a single node, or $\lceil \lg N \rceil$; if N is odd, it is $\lceil \lg N \rceil + 1$ [17].

Farley and Proskurowski [10] have carried out an extensive analysis of the gossip problem under the F1 model for rings and two-dimensional grids, toroidal grids, and Illiac grids. Their results include the following. For the N-node ring the minimum time is $N/2$ (the diameter) if N is even; if N is odd it is $\lceil N/2 \rceil + 1$ (two more than the diameter). The minimum time for any two-dimensional grid other than the 3×3 grid is the diameter of the grid; for the 3×3 it is 5 (one more than the diameter). The minimum time for an $N \times M$ toroidal grid is equal to the diameter if N and M are even, is between 1 and 2 greater than the diameter if just one of N and M is odd, and is between 2 and 4 greater than the diameter if both are odd. Similar results are

obtained for the Illiac grid.

For d-dimensional grids and toroidal grids in general, their results indicate that the minimum time is no greater than $2d$ plus the diameter. Whether it is possible to do better than that is an open question; for example, it is not known whether the $3 \times 3 \times 3$ grid can be solved in six steps under the F1 model.

3.3. Half-duplex simultaneous communication (H$*$). For any bipartite graph with diameter D under the H$*$ model, there is a $(D+1)$-step solution to the gossip problem: two-color the nodes red and black and have red nodes transmit to all neighbors on even time steps and black nodes do so on odd time steps.

For the complete graph, it takes exactly two steps to solve the problem regardless of the size of the graph. It is impossible to solve the problem in one step, and for a two-step solution we just select a subgraph consisting of one node and one edge from it to every other node and apply the above construction with the first transmissions going toward the central node.

Since the d-dimensional hypercube is bipartite, the above bipartite approach yields a $(d+1)$-step solution. For $d \leq 2$ this is optimal, but for $d \geq 3$, solutions in d steps exist. They are hard to find and describe—for example, there are 2^{36} different three-step strategies to consider on a 3-cube—and we treat them in a separate paper [18]. One significant result of that work is that for all $d \geq 4$, time-invariant optimal solutions, that is, solutions where the transmissions are the same at each step exist.

For the d-dimensional grid of diameter D, §7 shows that the problem can be solved in D steps even under the pairwise model, and this automatically provides an optimal solution under the simultaneous model.

A ring with an even number of vertices N is bipartite, and the above bipartite algorithm gives an optimal solution of $(N/2+1)$ steps, one step more than the diameter. (It is impossible to do it in $N/2$ steps.) With an odd number of vertices, the following modified version can be used. On each step, one link is omitted from involvement, the choice of which link to omit proceeding around the ring one step at a time. The other links participate in the basic bipartite pattern, where directions are chosen so that each node alternates sending and receiving. This yields a solution taking $(N-1)/2+2$ steps, two steps more than the diameter.

For the d-dimensional toroidal grid the same construction can be used. In each dimension where the size is odd, the pattern must be modified as with the ring, where the omitted links are chosen with matching coordinates so that they form entire rows or columns (or hyperplanes). This yields solutions that take $(D+k+1)$ steps, where k is the number of dimensions where the size is odd.

The rest of this paper is devoted to the half-duplex pairwise (H1) model, the analysis of which yields a surprising richness of nontrivial and interesting problems and results.

4. Complexity of finding optimal solutions. In this section we demonstrate that the problem of finding time-optimal algorithms for gossiping is NP-complete under either the H1 or the F1 model of communication. This result serves to justify the detail required to find optimal algorithms for specific interconnections. Our reduction uses the Minimum Broadcast Time Problem as described by Garey and Johnson [11] giving the result for the H1 model, which we then show can be modified to yield the same result for F1. We also need a simple result about reversibility when using the H1 model. First, we restate the Minimum Broadcast Time problem from [11].

Minimum Broadcast Time Problem (MBTP): Given a graph $G = (V, E)$, a subset V_0 of V, and a positive integer K, can a message that is originally resident on all nodes in V_0 be distributed to all of V in K steps using the H1 model?

This formulation uses our terminology instead of the set and edge formulation in [11] but the reader can easily verify that the two are precisely the same. The result is NP-complete even for $|V_0| = 1$, that is, for broadcasting from a single node. Observe that any single source broadcast algorithm becomes a single sink gather algorithm when the sequence of communications is reversed. That is, by running a broadcast algorithm backwards, we collect tokens from all nodes.

Given an instance of the MBTP involving a graph $G = (V, E)$ and a specified node v, we construct another graph $G^* = (V^*, E^*)$ that depends on the choice of v with the property that the MBTP for G and v has a solution using k or fewer steps if and only if the gossip problem for G^* has a solution using $2k + 2$ or fewer steps.

The graph G^* is constructed as follows. Take two copies of G and connect the nodes corresponding to v from each copy. This is G^*. We denote the two versions of G by G_1 and G_2 and the two corresponding versions of v by v_1 and v_2.

Suppose that we have a solution to the MBTP for G and v that uses k steps. Run the algorithm in reverse on each copy of G in G^*. That takes k steps. Now all tokens from G_1 are at v_1 and all tokens from G_2 are at v_2. It takes two steps to exchange the sets between v_1 and v_2 in the H1 model. We then run the MBT algorithm on each copy of G from v_1 and v_2. This solves the gossip problem in $2k + 2$ steps.

Now assume that we have a solution to the gossip problem that uses $s \leq 2k + 2$ steps. Let t be the step after which the last token from G_1 first reaches v_2. Since the last token reached v_2 after t steps, all other tokens have already reached v_2. Moreover, all the tokens must have first reached v_1 by time $t - 1$ since it takes one time step to cross from G_1 to G_2. Since this is a solution to the gossip problem using s steps, that last token to reach v_2 after t steps must be broadcast to all of G_2 in $s - t$ steps. Hence we have two MBTP candidate algorithms: the gather algorithm that used $t - 1$ steps to gather all tokens at v_1 (when run in reverse, this would be a broadcast algorithm for G) and the broadcast algorithm from v_2 to all of G_2 that uses $s - t$ steps. Since $((s - t) + (t - 1))/2 = (s - 1)/2$, one of the algorithms must use no more than $\lceil (s - 1)/2 \rceil \leq k$ steps.

Thus a solution to the MBTP for G and v in k steps exists if and only if there is a solution to the gossip problem for G^* in $2k + 2$ steps. Since the construction of G^* uses only polynomially much resources, we have shown that the minimum-time gossip problem is NP-complete.

Finally, we observe that in broadcasting from a single node (or collecting to a single node), the full-duplex capability is useless: there can be no case where the replacement of a half-duplex transmission by a full-duplex one would improve any of the above algorithms. Hence the entire above proof can be repeated for the F1 model, with $2k+1$ being used in place of $2k+2$ because the exchange between v_1 and v_2 can be done in one step.

5. The complete graph. Analysis of the gossip problem for the complete graph is important because it provides a benchmark for all other graphs. Entringer and Slater [8] have presented an algorithm that provides an upper bound, and we show that this algorithm is optimal by deriving a matching lower bound. After presenting this derivation, we present their algorithm and then discuss some implications for parallel algorithms.

5.1. Lower bound. We develop a lower bound by introducing a measure of aggregation and showing that there are limits on the rate at which that measure can grow. Given the complete graph with $N = 2n$ vertices $v_1, v_2, \cdots, v_{2n}$, let us denote by $x_i(t)$ the number of distinct tokens present at v_i at any time t. Writing these values as a vector $\mathbf{x} = (x_1, x_2, \cdots, x_{2n})$ we can use as our measure the norm $l_p(\mathbf{x}) = \left(\sum_1^{2n} x_i^p\right)^{1/p}$ for some suitable $1 \leq p \leq \infty$. (For $p = \infty$, $l_p(\mathbf{x}) = \max_1^{2n} x_i$.) We will choose p so as to maximize the resulting lower bound on the number of steps required in the complete algorithm.

Let us denote by $m_p(t)$ the maximum value that the measure l_p can possibly achieve after t steps and seek a relationship between $m_p(t)$ and $m_p(t+1)$. Our derivation is based on the following formula.

LEMMA 5.1. *If $A_1, A_2, \cdots, A_s, B_1, B_2, \cdots, B_s$ are positive real numbers, then*

$$\frac{A_1 + A_2 + \cdots + A_s}{B_1 + B_2 + \cdots + B_s} \leq \max_i \frac{A_i}{B_i}.$$

Proof.

$$A_j \leq B_j \max_i \frac{A_i}{B_i} \Longrightarrow \sum_j A_j \leq \left(\sum_j B_j\right) \max_i \frac{A_i}{B_i} \Longrightarrow \frac{\sum_j A_j}{\sum_j B_j} \leq \max_i \frac{A_i}{B_i}. \qquad \square$$

Step t consists of pairwise communications; assume that the number of tokens present at the senders and receivers are $s_1, s_2, \cdots, s_n$ and $r_1, r_2, \cdots, r_n$, respectively. If M denotes the maximal value of $\frac{m_p(t+1)}{m_p(t)}$, we have

$$M^p = \max_{1 \leq s_1, r_1, s_2, \cdots, r_n} \frac{s_1^p + (s_1 + r_1)^p + \cdots + s_n^p + (s_n + r_n)^p}{s_1^p + r_1^p + \cdots + s_n^p + r_n^p}.$$

Now we apply Lemma 5.1 with $A_j = s_j^p + (s_j + r_j)^p$ and $B_j = s_j^p + r_j^p$, and at the same time observe that the maximum is achieved when $s_1 = s_2 = \cdots = s_n$ and $r_1 = r_2 = \cdots = r_n$, so that we obtain

$$M^p = \max_{1 \leq s, r} \frac{s^p + (s+r)^p}{s^p + r^p}.$$

Now before considering our choice of p, let us see what the measure says about how many steps the token exchange will take. Initially, each node has just one token, so $m_p(0) = N^{1/p}$. Upon completion at time T, each node will have N tokens, so that $m_p(T) = N^{(p+1)/p}$. By definition of M, we must have $m_p(T) \leq M^T m_p(0)$, which yields the following lower bound:

$$T \geq \frac{\lg N}{\lg M}.$$

Now to obtain the best possible lower bound with this approach, we choose p to maximize T in the above equation. This means that we want to minimize M. Now define

$$V(p) = \max_{(r,s) \neq (0,0)} \left(\frac{|s|^p + |s+r|^p}{|s|^p + |r|^p}\right)^{1/p}.$$

It is immediate that $M \le V(p)$, so $\min_p M \le \min_p V(p)$; it is the latter quantity that we can find using some properties of vector and matrix norms. (It turns out that $\min_p M = \min_p V(p)$.) Consider the linear operator

$$F\begin{pmatrix} s \\ r \end{pmatrix} = \begin{pmatrix} s \\ s+r \end{pmatrix},$$

or in matrix notation,

$$F\begin{pmatrix} s \\ r \end{pmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}\begin{bmatrix} s \\ r \end{bmatrix}.$$

Denote the matrix representing F in this basis by F also. We need some standard norm definitions at this point (see [13] for more about the definitions and properties of these norms). Define the vector p-norm according to:

$$\|x\|_p = (|s|^p + |r|^p)^{1/p} \quad \text{where } x = \begin{pmatrix} s \\ r \end{pmatrix}.$$

A fundamental fact about vector p-norms is that for $1 \le p < \infty$ and $1/p + 1/q = 1$, we have

$$\|x\|_p = \max_{\|y\|_q = 1} y^{\mathrm{T}} x = \max_{y \ne 0} \frac{y^{\mathrm{T}} x}{\|y\|_q},$$

where y^{T} denotes transposition. For a matrix G the matrix p-norm of G is defined as

$$\|G\|_p = \max_{x \ne 0} \frac{\|Gx\|_p}{\|x\|_p}.$$

Now, note that

$$\|F\|_p = V(p)$$

by these definitions. By a classical theorem of Riesz, $\log \|G\|_p$ is a convex function of p [26, p. 179]. Moreover,

$$\begin{aligned}
\|F\|_p &= \max_{\|x\|_p = 1} \|Fx\|_p \\
&= \max_{\|y\|_q = 1} \max_{\|x\|_p = 1} y^{\mathrm{T}} F x \\
&= \max_{\|x\|_p = 1} \max_{\|y\|_q = 1} x^{\mathrm{T}} F^{\mathrm{T}} y \\
&= \|F^{\mathrm{T}}\|_q.
\end{aligned}$$

But F and F^{T} clearly have the same norms since, apart from a basis permutation, they are the same operator. Thus, for $1/p + 1/q = 1$, we have $V(p) = V(q)$. Using this and the fact that $\log V(p)$ is convex as a function of p, we see that the minimum value of $V(p)$ is attained for $p = 2$.

Now we seek the value $V(2)$. Observe that the maximum cannot occur for $s = a$ (compare, for example, $s = 0, r = 1$ with $s = r = 1$), so that we can set $z = r/s$ and write

$$V(2) = \max_z \left(\frac{1 + (1+z)^2}{1 + z^2} \right)^{1/2}.$$

Using elementary calculus one can now show that the maximum occurs for $z = \rho - 1$, where ρ is the golden ratio. Evaluating the formula at that point yields

$$T \geq \frac{\lg N}{\lg \sqrt{(1+\rho^2)/(1+(\rho-1)^2)}} = \frac{\lg N}{\lg \rho} \approx 1.44 \lg(N) = \log_\rho N.$$

Thus we have the following result.

THEOREM 5.2. *The gossip problem for the complete graph under the* H1 *model of communication requires at least* $\lceil \log_\rho N \rceil \approx 1.44 \lg N$ *steps, where* ρ *is the golden ratio* $(1+\sqrt{5})/2$.

5.2. An optimal algorithm. In this section we present in our own notation an optimal algorithm for the gossip problem under the H1 model which was described by Entringer and Slater [8]. For completeness, we restate the proof of its performance. The algorithm uses the *Fibonacci* sequence, which we define by $F_0 = 0$, $F_1 = 1$, $F_{k+1} = F_k + F_{k-1}$ for $k > 1$.

We assume N is even; if not, choose any one node and send its token to another site, solve the problem on all nodes but the one, and lastly, send from some site to the chosen node. Let $n = N/2$ and label the nodes $p_0, p_1, \cdots, p_{n-1}, q_0, q_1, \cdots, q_{n-1}$. Denote by P_i (Q_i) the token originating at p_i (q_i). Transmit as follows until the token exchange is complete:

Step 0: Transmit from p_i to q_i for all i, $0 \leq i \leq (n-1)$.
Step 1: Transmit from q_i to p_i for all i, $0 \leq i \leq (n-1)$.
Step k, $k \geq 2$: If k is even transmit from p_i to $q_{i+F_{k-1}}$ for all i, where $+$ denotes addition modulo n. If k is odd, transmit similarly from q_i to $p_{i+F_{k-1}}$ for all i.

LEMMA 5.3. *After step* $k \geq 2$, *where* $F_{k+1} \leq n$, *if* k *is even* (*odd*), *for each* i, q_i (p_i) *has the tokens* $P_i, Q_i, P_{i-1}, Q_{i-1}, \cdots, P_{i-F_{k+1}+1}, Q_{i-F_{k+1}+1}$, *and* p_i (q_i) *has the tokens* $P_i, Q_i, P_{i-1}, Q_{i-1}, \cdots, P_{i-F_k+1}, Q_{i-F_k+1}$.

Proof. This is easily proved by induction on k. □

THEOREM 5.4. *The gossip problem for the complete graph under the* H1 *model of communication can be solved in* $\lceil \log_\rho N \rceil + 4$ *steps.*

Proof. From the above lemma, it is clear that after step k, nodes p_i and q_i (k even) or nodes q_i and p_i (k odd) have $2F_k$ and $2F_{k+1}$ tokens, respectively. Then it can be seen that when $F_k \geq n$, every node has all tokens and the broadcast is finished. Now the result follows from the well-known inequality $F_k \geq \rho^{k-2}$. □

This algorithm asymptotically takes exactly the same number of steps given by the lower bound discussed in the preceding section. In fact it achieves at each step very close to the maximum rate of growth of information as we defined it in establishing the lower bound, and it solves the broadcast problem efficiently (usually within 1 step of the lower bound) for all even values of N including small ones.

5.3. Lower bounds for specific problems. As mentioned in the introduction, the gossip problem provides a lower bound on the communication complexity of algorithms for which all outputs require all inputs. Specifically, suppose we have an arbitrary interconnection network and we have data distributed over N_1 nodes, say, d_i at node v_i for $i = 1, \cdots, N_1$. Furthermore, suppose that we desire to compute functions f_j at nodes w_j for $j = 1, \cdots, N_2$. These functions are such that $f_j(d_1, \cdots, d_{N_1})$ depends explicitly on all data d_i. Thus in terms of tokens, we can say that all tokens at the input nodes must be distributed to all output nodes.

Following the idea in the proof of Theorem 5.2, note that the initial value of the aggregate information measure is $m_2(0) = N_1^{1/2}$ and the final aggregate information measure must be at least $N_1 N_2^{1/2}$. It can be readily verified that the construction leading up to Theorem 5.2 applies to this case as well, so that a lower bound on the number of communication steps required to compute the functions is given by

$$\begin{aligned} T &\geq \frac{\lg((N_1 N_2)^{1/2})}{\lg M} \\ &= \frac{(\lg N_1 + \lg N_2)}{2 \lg \rho}, \end{aligned}$$

where ρ is the golden ratio as before. Note that we made no assumptions about whether some or all of the input nodes were the same as the output nodes; a slightly tighter bound can be derived if the input and output nodes are distinct.

This result is interesting to juxtapose against some well-known parallel algorithms. Consider first the computation of the Discrete Fourier Transform on N points. The Discrete Fourier Transform itself consists of N outputs, each one of which depends explicitly on each input. Thus the above result states that at least $1.44 \lg N$ communication steps are required for the computation of the transform using any algorithm and any architecture, providing of course that the data is distributed as above. By comparison, the standard approach to parallelizing the Fast Fourier Transform requires $2 \lg N$ communication steps. Thus from a communications complexity point of view, the Fast Fourier Transform is a constant factor of about $\frac{1}{3}$ worse than the optimal communications algorithm would be. The optimal algorithm from a communications point of view would be the optimal token exchange algorithm described above for the complete graph followed by a brute force Discrete Fourier Transform calculation at each output node.

A similar result can be stated for matrix inversion. Suppose that the n^2 entries of a matrix are distributed over n^2 processors. Suppose that we wish to compute the inverse of the matrix with the entries of the inverse distributed over the n^2 processors likewise. Since the entry of every element of the inverse depends on every element of the original matrix, this leads to a gossip problem on n^2 processors. It has been known for some time [7], [21] that the inverse is computable in $O(\lg^2 n)$ parallel computation steps. Our results show that at least $2.88 \lg n$ communication steps are required.

It is clearly possible to obtain bounds on the communication complexity of a large class of problems for which input data and output are distributed across the nodes of a distributed memory multiprocessor.

6. The hypercube. The hypercube is the most difficult of the graphs for which we have analyzed the gossip problem. Its importance derives from the fact that it is the topology used in the most commercially successful and widely available multiprocessor computers. We believe the main reason for the difficulty is its low diameter: for graphs whose diameter is large relative to their size, it is usually possible to solve the problem in a number of steps close to the diameter. Since the diameter is a trivial lower bound, tight upper and lower bounds are typically derivable. However, for graphs with small diameters, relatively fast solutions are possible in principle, but it becomes difficult to either find them or find a nontrivial lower bound. The analysis for the complete graph follows this pattern, while the hypercube proves to be even more difficult.

The best known upper bound for the hypercube using the H1 model follows from an algorithm presented in another paper [18]. That algorithm is quite complex and so is not included in this paper, but we will summarize the results.

We note that there is a large class of algorithms solving the gossip problem on a d-dimensional hypercube in $2d$ steps. This class of algorithms has been known to many hypercube researchers (see, for example, [28], [23], [27]) and we only summarize it here. Let π_{i-} (π_{i+}) denote the parallel transmission of tokens from nodes with their ith bits cleared (set) to their neighbors with their ith bits set (cleared). Let π be the composition of these $2d$ unique transmissions in any order. The claim is that every such π solves the gossip problem in precisely $2d$ steps.

This is easy to demonstrate by picking any two nodes from the hypercube, say, α and β. These two labels differ in some number of bits and we obtain β's label from α by toggling no more than d of α's bits. Now π effectively involves sending tokens along each direction of each communications channel in the hypercube at some time, which can be viewed as toggling bits in labels. Hence some subsequence of transmissions will result in toggling the bits of α in such a way that β's label is obtained.

Furthermore, if π is composed of any $2d-1$ or fewer of these transmissions in any order, then π does not perform a complete token exchange. This is because, given an omitted π_{i-} or π_{i+}, one simply looks at the α and β that require the toggling of the bit of α that this omitted transmission corresponds to.

It is tempting to believe that this class of algorithms is optimal for hypercubes. (This is conjectured in [4].) Surprisingly, it is not. We know from the complete graph case that the lower bound for a hypercube is at least $1.44d$ while the above construction demonstrates the existence of algorithms requiring $2d$ steps. In [18], an algorithm for performing token exchange on a nine-dimensional hypercube in only 17 steps is presented showing that $2d$ is not in fact optimal for all cubes. The nine-dimensional cube is the smallest cube for which an algorithm using fewer than $2d$ steps is known. That algorithm generalizes for larger-dimensioned cubes giving an upper bound of $1.88d$ steps and this is the best known for hypercubes. This gap between $1.44d$ and $1.88d$ leaves plenty of room for future improvement, but at present we have no concrete ideas about how to proceed nor do we have strong intuition about which direction is most fruitful to pursue (that is, whether it is easier to raise the lower bound or to lower the upper bound).

7. The grid. Multidimensional grids have relatively large diameters, and this makes it relatively easy to solve the gossip problem optimally (in a number of steps equal to the diameter of the grid). Farley and Proskurowski [10] showed that two-dimensional grids other than 3×3 are solvable in optimal time under the F1 model. Recently, Bagchi, Hakimi, Mitchem, and Schmeichel [4] have shown that for $n \times m$ grids under the H1 model where $n \geq 6$ and m is even (odd), gossip is possible in one step (two steps) more than the diameter. They also showed that if $m = 2$, then gossip in optimal time is impossible.

In this section we sharpen those results by showing that optimal solutions exist for all sufficiently large two-dimensional grids, and we extend this result to grids in an arbitrary number of dimensions. The time to gossip under the H1 model on small grids is left open, although we are able to show that optimal solutions do not exist when a multidimensional grid's size is 2 or 3 in any dimension.

Figure 1(a) shows a communication pattern for the 5×5 grid. The numbers alongside the arrows indicate on which step(s) of the algorithm the indicated communication steps occur. The large solid dots will be called *primary* points and the hollow dots *secondary* points. The reader can verify that tokens present at the primary points are distributed to all primary points in optimal time. A further property that will be used in the case of three-dimensional grids is that any token present at a

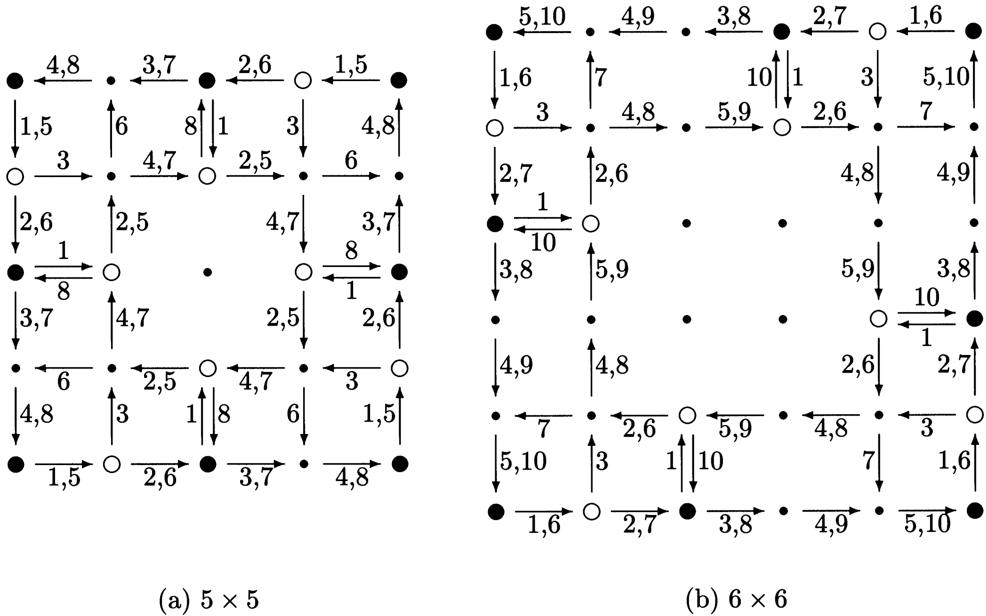


FIG. 1. *Distribution among primary points accomplished in optimal time.*

secondary point after the second step is distributed to all primary points by the end. Similar patterns with these two properties can be constructed for the 6×6 grid, as shown in Fig. 1(b), and for the 5×6 grid (not shown).

Given, then, a 9×9 or larger two-dimensional grid, if its size is odd in both dimensions we select a 5×5 subgrid exactly centered in the larger grid. If its size is even in one or two dimensions, we select a centered 5×6 or 6×6 subgrid. The reader can easily verify that if K is the distance from a corner of the larger grid to a corner of the subgrid, it is possible to simultaneously send each token on the grid to some primary point in K steps. (This is easier to do on larger grids; the 9×9 is the smallest grid where it is easy to do. Observe that each corner point of the subgrid can collect tokens from a 4×4 corner region of the larger grid in four steps.) To solve the token exchange on the full grid, then, we concentrate all tokens at the primary points, apply the communication pattern of Fig. 1, and then distribute tokens from the primary points using the reverse of the pattern that was used to concentrate the tokens. This gives a solution for any two-dimensional grid containing a 9×9 grid in a number of steps equal to the diameter.

We can extend this result to higher dimensions. Suppose we are given an $n_1 \times n_2 \times \cdots \times n_d$ grid with all $n_i \geq 9$. By holding all but the first two coordinates fixed we define $n_3 n_4 \cdots n_d$ independent two-dimensional grids. In parallel, we solve the problem on those grids in optimal time. Then we hold all but the third and fourth coordinates fixed and solve the problem optimally on those grids, and so on. If d is even we are finished after solving $d/2$ two-dimensional problems, while if d is odd we must at the end solve a three-dimensional version of the problem.

We can solve the three-dimensional version of the problem in optimal time on any grid large enough to contain a $9 \times 9 \times 5$ subgrid. For an $n_1 \times n_2 \times n_3$ grid, we first hold the third coordinate fixed and concentrate tokens at the primary points of the

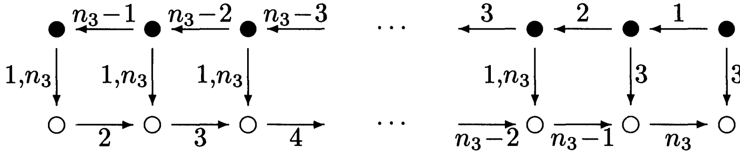


FIG. 2. *Distribution to secondary points throughout the third dimension.*

central subgrids of all the $n_1 \times n_2$ grids. Now for each pair consisting of a primary and associated secondary point, consider the $2 \times n_3$ subgrid defined by selecting that pair using all possible values of the third coordinate. After collecting tokens at the primary points, we apply the n_3-step pattern depicted in Fig. 2 on all these grids; we use this in place of the first step of Fig. 1. This sequence carries tokens from primary to associated secondary points while distributing the tokens throughout the third dimension. Lastly, we apply the remainder of the two-dimensional strategy on all the $n_1 \times n_2$ grids. Since the communication pattern in Fig. 1 takes tokens present at any secondary point to all primary points, we have distributed tokens throughout the three-dimensional grid in optimal time. We summarize these results in the following theorem.

THEOREM 7.1. *Assuming that $d \geq 2$ and $n_i \geq 9$ for all i, the gossip problem for the $n_1 \times \cdots \times n_d$ grid under the* H1 *model of communication is solvable in a number of steps equal to the diameter of the grid.*

For small multidimensional grids the minimum time to gossip is still open. We can show that the grid's size cannot be 2 in any dimension if the problem is to be solved in optimal time. Assume we have an optimal solution for a multidimensional grid of diameter D whose size in some dimension is 2. Consider neighbors p and q at a corner: points at a distance D from opposite points $\bar{p}$ and $\bar{q}$. The token from p must reach $\bar{p}$ in D steps, and thus it must be advanced toward that goal on every step so that after k steps there is a unique point $P(k)$ at distance k from p that has received the token. Similarly, there is a unique point $Q(k)$ at distance k from q that has received the token from q after k steps. Now let K be the unique time such that the token from q is not present at $P(K)$ after step K and is present at $P(K+1)$ after step $K+1$. Since step $K+1$ is a transmission from $P(K)$ to $P(K+1)$, the token from q must be present at $P(K+1)$ after step K. Since $P(K+1)$ is at distance $K+1$ from p, it is at distance K or more from q; the fact that the token from q is present there after K steps means that $P(K+1) = Q(K)$. But this is a contradiction since the token from q is not advanced from $Q(K)$ on step $K+1$.

A similar argument can be used with a grid of diameter D whose size in some dimension is 3. Let p, q, and r be points in a corner with p at distance D from $\bar{p}$, q at distance D from $\bar{q}$, and r in between p and q. Define $P(k)$ and $Q(k)$ as before. Let K_p be the unique time such that the token from r is not present at $P(K_p)$ after step K_p and is present at $P(K_p+1)$ after step K_p+1, and let K_q be the unique time such that the token from r is not present at $Q(K_q)$ after step K_q and is present at $Q(K_q+1)$ after step K_q+1. Reasoning as before, we determine that the token from r travels distance K_p in optimal time to $P(K_p+1)$, from which point it does not advance on the next step; and it travels distance K_q in optimal time to $P(K_q+1)$, from which point it does not advance on the next step. Since $P(K_p+1)$ and $P(K_q+1)$ must be different points, this is impossible.

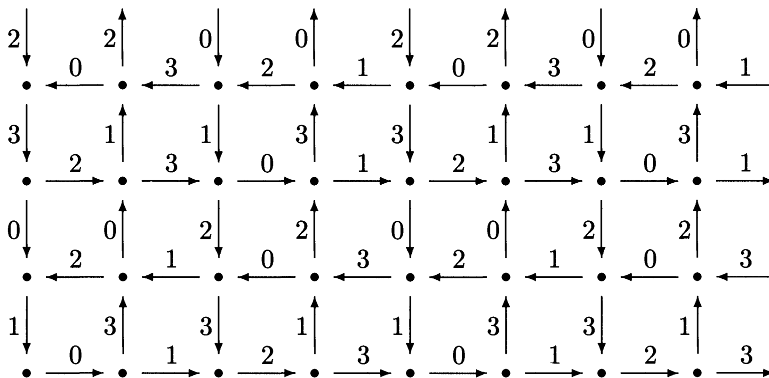


FIG. 3. *The basic pattern for the toroidal grid.*

We have thus determined that for the gossip problem under H1 to be solved in optimal time, the grid's size in each dimension must be at least 4. In two dimensions we have found optimal-time solutions for 7×7 and larger grids, which we do not show because of space limitations. (See [20].) This leaves open the question of optimal-time solutions for grids between 4×4 and $6 \times n$. In three dimensions the smallest optimal-time solution we know is the one described above for the $9 \times 9 \times 5$ grid.

8. The toroidal grid. The toroidal grid is obtained from the regular grid by adding connections that "wrap around" to connect the end points in each dimension. Formally, an $n_1 \times n_2 \times \cdots \times n_d$ toroidal grid has as vertices the points $(p_1, \cdots, p_d)$ in d-dimensional Euclidean space where $0 \leq p_i < n_i$ for all i, and where there is an edge connecting two points if their coordinates differ in just one component i and that difference is ± 1 modulo n_i. The diameter of the toroidal grid is half that of the regular grid, making it much harder to solve the gossip problem in a number of steps close to the diameter. We show that for $d \geq 2$ the gossip problem on a d-dimensional toroidal grid with diameter D is solvable in $D + 18d + 39$ steps. (If $d = 1$ then we have the degenerate case of a ring that is quite different in nature and that is taken up in §9.) First we sketch the strategy for the case $d = 2$, and then we present a general version for $d \geq 2$. These strategies are best suited for grids whose sizes in all dimensions are divisible by four while adjustments must be made for other sizes.

8.1. The algorithm for two dimensions. The basic pattern for two dimensions is shown in Fig. 3. It uses a cycle of four steps that is repeated indefinitely. Arcs that have transmissions occurring on the first step of the cycle are labeled with 0, those with transmissions on the second step 1, and so on.

The following properties are evident: (1) Tokens move at maximal speed in both horizontal and vertical directions. (2) Horizontal lines alternate between right-going and left-going directions, and vertical lines alternate between up-going and down-going. (3) A token traveling in one direction will, with a delay of two steps, be picked up at any node and also begin traveling in the orthogonal direction from that node. (4) The pattern corresponds to a synchronization of stoplights in a grid of alternating one-way city streets so that traffic can move forward at constant speed on all streets simultaneously without ever having to stop at a red light. (The downtown of Tulsa, Oklahoma has just such an arrangement.)

The optimality of the strategy is apparent from the following observations. Given

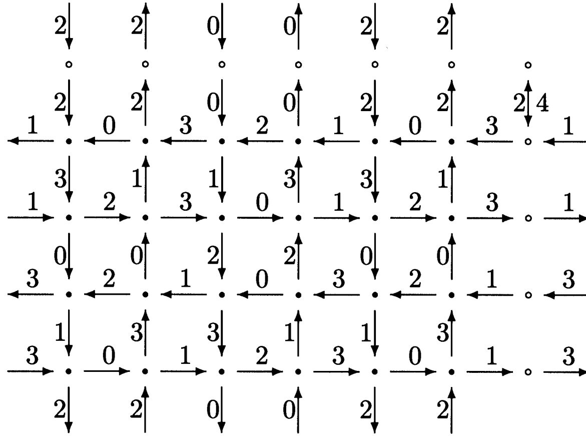


FIG. 4. *The solution for a 7 × 5 toroidal grid. Hollow points represent emulator and extra points. Apparently conflicting vertical transmissions at the top are carried out in alternating cycles.*

some source node and some target node, the token from the source will, within some constant number of steps, arrive at a nearby location where the horizontal line is oriented toward the target. Then it can travel along that horizontal line at a rate of one step per time unit until it reaches a vertical line that is oriented toward the target and that passes within one unit of the target. Then, after a delay of 2, it can proceed along this vertical line at the rate of one unit per time step until it arrives at a point near the target from which it will reach the target within some constant number of final steps. Thus the token from an arbitrary source reaches an arbitrary target in a number of steps equal to the distance between them plus some constant that is independent of the two-dimensional grid size.

This strategy is most easily implemented on a grid whose horizontal and vertical sizes are multiples of 4, in which case one can verify that the solution takes at most 18 steps more than the diameter of the grid. The strategy can be modified to handle the other cases, as can be seen in Fig. 4 for the 7×5 toroidal grid. Roughly speaking, the above strategy is used on as many nodes as possible, leaving at most one line of nodes in each dimension that violate the consistency of the sequence of transmissions. Those extra nodes then perform only a subset of transmissions in a way that is consistent with neighboring nodes' transmission sequences. The reader is encouraged to generalize the situation depicted in Fig. 4, while more details can be found in the longer technical report [20].

8.2. A strategy for arbitrary dimensions. One would hope that toroidal grids in 3, 4, $\cdots$ dimensions (with all sizes divisible by 6, 8, $\cdots$) could be handled by transmission patterns with periods of 6, 8, $\cdots$ that propagate tokens at maximal speed along lines in all directions, directly generalizing the two-dimensional strategy. Efforts at constructing such strategies have convinced us that no such pattern exists in three dimensions. Our strategy for grids of larger dimension generalizes the two-dimensional version in the following, weaker way. We retain the same basic four-step pattern and we specify transmissions in the horizontal directions exactly as before. (We base this on a node's horizontal coordinate coupled with its *distance* from the horizontal axis; this specification is formalized in the function T below.) In place of

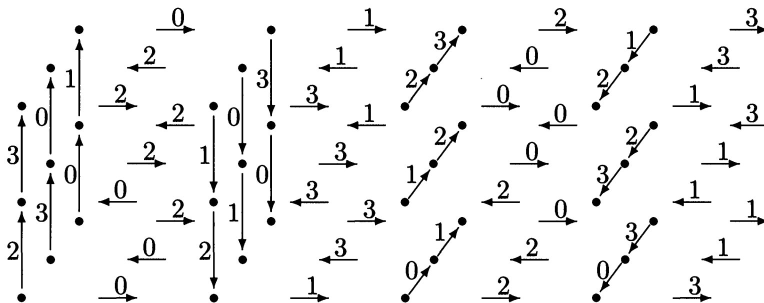


FIG. 5. *The basic pattern for the three-dimensional toroidal grid.*

the vertical lines, we consider hyperplanes orthogonal to the horizontal lines. In each hyperplane we choose one direction (forward or backward through some dimension) and provide transmissions in that direction in the same way as in the two-dimensional version. We choose these dimensions (using function U below) and directions (with forward and backward pairs) so that as one proceeds along a horizontal line, one passes all directions and dimensions in a cyclic pattern. Figure 5 shows the pattern for $d = 3$. The following description presents the construction precisely, and it includes the special treatment necessary for sizes that are not multiples of four, treatment which differs from what was used in the two-dimensional case for sizes not divisible by four.

We begin with a toroidal $n_1 \times \cdots \times n_d$ grid with $d \geq 2$, $n_1 \geq 8d$, and all other $n_i \geq 4$. For each i let m_i be the largest integer less than or equal to n_i that is divisible by 4. We shall divide the points of the grid into three groups: a point $p = (p_1, \cdots, p_d)$ is a *regular* point if $p_i < m_i$ for all i; it is an *emulator* point if $p_i \geq m_i$ for exactly one i; otherwise ($p_i \geq m_i$ for two or more values of i) it is an *extra* point. The regular points form a $m_1 \times \cdots \times m_d$ subset of the grid that is the largest one that has a size divisible by 4 in every dimension. The emulator points lie outside this subgrid on the lines occupied by the regular points. In general, there may be no emulator and/or extra points for a particular grid; our constructions work in those cases just as well (faster, in fact) as when the emulator and extra points are present.

Let U be any bijection from $\{0, 1, \cdots, d-2\}$ to $\{2, 3, \cdots, d\}$. Now define functions Q and T:

$$Q(y, z) = z + y(-1)^z \pmod 4,$$
$$T(p) = Q(p_1, p_2 + \cdots + p_d).$$

These two facts about Q are used in the first proof below: $Q(y+1, z) = Q(y, z) + (-1)^z$ and $Q(y, z+1) = Q(y, z) + (-1)^y$.

We now define *Strategy* G for the regular and emulator points. Each regular point transmits alternately through dimension 1 (called the horizontal direction) and some other dimension (called the orthogonal direction), with the exact choices determined by the functions T and U. Specifically, regular point p transmits through dimension 1 at all time steps that are congruent to $T(p)$ modulo 4; the transmissions go in the forward direction (toward increasing values of p_1) if $p_2 + \cdots + p_d$ is even and they go in the reverse direction if it is odd. At time steps congruent to $T(p) + 2$, p transmits

in a way determined by p_1: if p_1 is even, p transmits in the forward direction through dimension

$$U\left(\frac{p_1}{2} \pmod{d-1}\right),$$

and if it is odd p transmits in the reverse direction through dimension

$$U\left(\frac{p_1-1}{2} \pmod{d-1}\right).$$

Emulator points, if there are any, occur in groups of one, two, or three along lines whose other points are regular points. If transmissions along those lines are defined for the regular points, we extend the transmissions to the emulator points as shown in Fig. 4. The result is that a token moving toward emulator points crosses from the last regular point before them to the first regular point after them in at most nine time steps. (In contrast, if there are no emulator points, this crossing is made in one time step, so there is an added delay of eight caused by the presence of the emulator points; this delay can also be expressed as seven more than the distance covered in the crossing.)

LEMMA 8.1. *Strategy G is well defined under the pairwise model of communication.*

Proof. All we must check is that when a point p transmits to its neighbor q, q is not supposed to be involved in any other transmission. We assume both p and q are regular points surrounded by regular points; when they or their neighbors are emulator or extra points, the proof is similar but easier. We treat the case where both p_1 and $p_2+\cdots+p_d$ are even; the other cases are similar. (a) For transmissions through dimension 1, observe that $q=(p_1+1, p_2+\cdots+p_d)$. Now note that $T(q)=T(p)+1$, and thus the strategy specifies transmissions away from q at times congruent modulo 4 to $T(p)+1$ and $T(p)+3$, which do not conflict with the transmissions from p to q at times congruent to $T(p)$. Regarding transmissions toward q, let us identify all nodes that transmit toward q: p is the only one that transmits toward q through dimension 1, and otherwise the dimension of a transmission to q is determined by p_1+1. Since all nodes with first coordinate p_1+1 transmit in the same direction, exactly one such node r transmits to q. Since p_1+1 is odd, the transmission is in the reverse direction and it occurs at times congruent modulo 4 to $T(r)+2=Q(p_1+1, p_2+\cdots+p_d+1)+2=T(q)+1=T(p)+2$, which causes no conflict with the transmissions from p to q at times congruent to $T(p)$. (b) For the transmissions from p at times congruent to $T(p)+2$ the analysis is similar. The transmissions go to a point q for which $T(q)=T(p)+1$. Point q transmits in the same direction at times congruent to $T(q)+2=T(p)+3$, transmits in the reverse direction in dimension 1 at times congruent to $T(q)=T(p)+1$, and receives from a point r in dimension 1 at times congruent to $T(r)=T(q)-1=T(p)$. None of these conflicts with the transmission from p to q at times congruent to $T(p)+2$. □

Now we proceed to solve the gossip problem using Strategy G. We distinguish between *central* regular points for which $2 \leq p_1 < m_1 - 2$ and *boundary* regular points, which are the others. Our solution consists of three phases: compression, Strategy G, and expansion. The compression phase consists of at most $2d+2$ steps in which we send all tokens from extra, emulator, and boundary regular points to central regular points: we simply treat each dimension in turn, and in one or two steps send each token toward its nearest regular point; for the horizontal dimension, we go two additional steps so that the tokens are sent to the central regular points.

In the second phase we perform a complete token exchange among the central regular points by letting Strategy G run for a suitable number of time steps. Finally, the expansion phase is the compression run in reverse and it distributes all tokens and the problem is solved. This leaves us with the problem of determining how Strategy G solves the token exchange among the central regular points.

We begin with a lemma that captures the essence of the solution. Let the distance between points x and y be denoted by $\delta(x, y)$.

LEMMA 8.2. *Suppose we have regular points p and r meeting either of the following conditions: horizontal transmissions at both p and r are in the forward (backward) direction, and in proceeding forward (backward) horizontally from p one encounters exactly $2d-2$ regular points, including the beginning and ending points, by the time the horizontal position of r is reached. (This means that if no emulator points intervene, p and r are separated by a horizontal distance of $2d-3$.) Then under Strategy G, if a token is present at p at time t where t is congruent to $T(p)$ (i.e., ready for a horizontal transmission away from p), then the token reaches r by time $t + 14d + 10 + \delta(p, r)$.*

Proof. Assume, without loss of generality, that horizontal transmissions at p go in the forward direction. We shall describe a path by which the token from p reaches r. The token generally proceeds forward horizontally. Each time it is located at a node where the transmissions in the orthogonal direction proceed *toward* r, we check whether the difference between p and r is even in that coordinate. If even, our path follows the orthogonal direction to the point where the coordinates match, and then it proceeds again in the horizontal direction; since the coordinates differ by an even amount, the path will continue again from there in the forward horizontal direction. If odd, we note the horizontal coordinate and let the path continue in the horizontal direction without following the orthogonal direction. When we encounter another such orthogonal direction representing an odd difference in coordinates, we have our path go in that direction until reaching the point where the coordinates match. From there, the transmissions in the horizontal dimension proceed backwards, and we follow them until reaching the horizontal coordinate that we previously noted. From there our path proceeds through the orthogonal direction at that point until reaching the point matching r in that coordinate. From this point, since we once again traveled an odd distance in the orthogonal direction, the horizontal transmissions will once again be proceeding in the forward direction, and we continue as at the beginning. The assumptions ensure that every possible direction will be encountered exactly once, and since there must be an even number of coordinates (other than the horizontal coordinate) where p and r differ by an odd amount—this is because the horizontal lines containing p and r are separated by an even distance—the token will eventually arrive at r.

Now let us count the number of time steps this process takes. The reader can verify that the slowest path involves the maximal amount of backward motion: three forward steps initially, then orthogonal motion, three backward steps, orthogonal motion, seven forward steps, orthogonal motion, three backward steps, orthogonal motion, seven forward steps, and so on. We can count steps by charging 0 for a transmission that reduces the distance to the target, 1 for a delay, and 2 for a transmission that increases the distance to the target, and then adding the distance between p and r. Each instance of orthogonal motion, then, is charged 4 for initial and final delays plus 7 for the fact that the orthogonal motion might traverse emulator points. Each instance of backward motion is charged 6. With $d-1$ instances of orthogonal motion and half that number of backward moves, the total charge is $14(d-1)$. The horizontal

motion might cross emulator points up to three times, and this adds up to 24 extra steps. Thus the time to get from p to r is at most $14d + 10 + \delta(p, r)$. □

Now we can establish the time Strategy G takes to exchange tokens among the central regular points. First observe that if we start at a central regular point p, with at most two horizontal steps we can proceed orthogonally one step to a regular point $R(p)$. This is true because at the starting point there might be an extra point in the place where an orthogonal transmission would be directed, and after one horizontal step the orthogonals might be in a different dimension and again an extra point be encountered, but by the second step the orthogonals would have to reverse and necessarily go to a regular point. Similarly, if q is a central regular point, then a regular point $\bar{R}(q)$ exists from which it is possible to reach q with an orthogonal step followed by 0, 1, or 2 horizontal steps.

Given central regular points p and q, we check the horizontal separation between p and q: we will follow the token from p in either the forward or the backward horizontal direction, according to which yields the shorter path to q. Without loss of generality, assume the forward direction is used.

Now if the horizontal distance between p and q is large enough, we can calculate the minimum time for the token from p to reach q as follows. If horizontal transmissions from p are in the backward direction, let $\hat{p} = R(p)$, otherwise let $\hat{p} = p$. It takes up to 10 steps to reach $\hat{p}$ ready for a horizontal transmission. If horizontal transmissions from q are in the backward direction, let $\hat{q} = \bar{R}(q)$, otherwise let $\hat{q} = q$. By Lemma 8.2, it takes up to $14d + 10 + \delta(\hat{p}, r)$ additional steps to reach the point r, which is on the same horizontal line as $\hat{q}$. Traveling from r to $\hat{q}$ forward horizontally incurs only a delay of two at the beginning—the possibility of crossing emulator points has already been accounted for in Lemma 8.2. Finally, it takes up to seven steps to reach q from $\hat{q}$. The total is then $14d+29+\delta(\hat{p}, \hat{q}) \leq 14d+35+\delta(p, q) \leq 14d+35+D$.

If the horizontal distance between p and q is not large enough that it is possible to travel, as described above, forward from r toward q, then the horizontal distance between p and q is at most $2d$, and using the fact that $n_1 \geq 8d$ one can show that the token from p reaches q in $14d + 33 + D$ steps. See [20] for details.

Combining the maximum time to distribute tokens among the central regular points with the $2d + 2$ steps for the initial compression and final expansion phases gives the following result.

THEOREM 8.3. *Assuming that $d \geq 2$, $n_1 \geq 8d$, and $n_i \geq 4$ for $i \geq 2$, the gossip problem for the $n_1 \times \cdots \times n_d$ toroidal grid under the* H1 *model of communication is solvable in $D + 18d + 39$ steps where D is the diameter of the grid.*

9. The ring. The ring with N vertices has diameter $\lfloor N/2 \rfloor$. Unlike the toroidal grid where tokens can be sent in different directions along different paths, tokens must travel both directions around the ring at the same time if we are to approach an $N/2$-step solution. Bagchi, Hakimi, Mitchem, and Schmeichel [4] have derived an upper bound of $N/2 + 3\sqrt{N}/2$ where N is the square of an even number, establishing an upper bound of $N/2 + O(\sqrt{N})$ in general. We sharpen this result by deriving a general upper bound of $\lfloor N/2 \rfloor + \left\lceil \sqrt{2N} \right\rceil + 2$ in §9.1 and a general lower bound of $\lfloor N/2 \rfloor + \left\lceil \sqrt{2N} \right\rceil - 1$ in §9.2.

9.1. Upper bound. Consider a ring with N vertices $v_0, v_1, \cdots, v_{N-1}$, and for convenience define $v_i = v_{i(\mathrm{mod} N)}$ for $i < 0$ and $i \geq N$. We will define values $0 \leq s_1 < s_2 < \cdots < s_P < N$, nearly evenly spaced around the ring and with ample space between them, and use them in this way. Initially we let both $v_{s_j} \to v_{s_j - 1}$ and

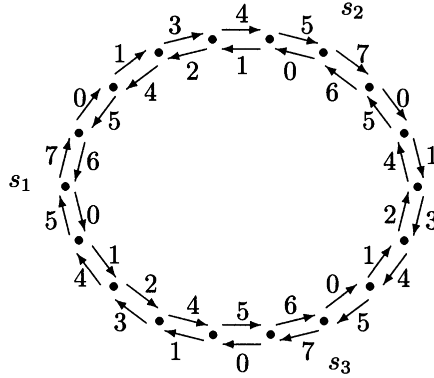


FIG. 6. *An optimal solution for the* 18*-node ring.*

$v_{s_j+1} \rightarrow v_{s_j+2}$ at $t = 0$, for all $1 \leq j \leq P$. For the rest of the strategy we *propagate* the transmissions in the following sense. If v_i receives from v_{i-1} at time t and if v_{i+1} and v_{i+2} were not involved in transmissions at time t, then at time $t+1$ let $v_i \rightarrow v_{i+1}$. Similarly, if v_i receives from v_{i+1} at time t and if v_{i-1} and v_{i-2} were not involved in transmissions at time t, then at time $t+1$ let $v_i \rightarrow v_{i-1}$. Otherwise we recognize two other possibilities: a *good crossing* where at time t, v_i receives from v_{i-1} and v_{i+1} receives from v_{i+2}, and a *bad crossing* where at time t, v_i receives from v_{i-1} and v_{i+2} receives from v_{i+3}. (In general, other cases might occur but they do not arise in our strategy.) For a good crossing we let $v_{i+1} \rightarrow v_i$ at time $t+1$, $v_i \rightarrow v_{i+1}$ at time $t+2$, and both $v_{i+1} \rightarrow v_{i+2}$ and $v_i \rightarrow v_{i-1}$ at time $t+3$. For a bad crossing we let $v_{i+2} \rightarrow v_{i+1}$ at time $t+1$, $v_{i+1} \rightarrow v_i$ at time $t+2$, $v_i \rightarrow v_{i+1}$ at time $t+3$, and both $v_{i+1} \rightarrow v_{i+2}$ and $v_i \rightarrow v_{i-1}$ at time $t+4$.

It can be seen that the above propagation technique carries groups of tokens forward and backward around the ring simultaneously, even when crossings occur. We visualize *buckets* moving forward and backward around the ring, picking up one token at each node and moving the accumulated collection forward. All buckets move at the same speed except that they are delayed when forward- and backward-moving buckets experience crossings. Backward-moving buckets are always delayed by one time step in a crossing, while forward-moving buckets are delayed one time step in a good crossing and two time steps in a bad crossing. The idea of the strategy is that each token should be carried to half the ring by a forward-moving bucket and to the other half by a backward-moving bucket. We will choose starting points spread out around the ring so as to minimize the time it takes to complete the algorithm. The spacing between forward-moving buckets is maintained during the algorithm, and so is the spacing between the backward-moving buckets, so that buckets moving in the same direction never overtake one another.

An example of this strategy for an 18-node ring is shown in Fig. 6. This ring happens to have an optimal solution that is symmetric, as shown in the figure. The solution is based on an eight-step cycle that repeats, completing the token exchange in 14 steps.

We first treat the case where N is even. Denote by Δ_j the gap $s_{j+1} - s_j$ between successive s_j values, where we for convenience define $s_{P+1} = s_1$. Suppose all values s_j and therefore Δ_j are even. In this case it is easy to verify that only good crossings

will ever occur. How long does it take the above given strategy to complete the token exchange? First choose some j and consider the locations v_f and v_b where $f = s_j + \Delta_j/2$ and $b = f + 1$. At time $t = \Delta_j/2 - 1$ the former is reached by the forward-moving bucket that started at v_{s_j+1} and the latter is reached by the backward-moving bucket that started at $v_{s_{j+1}}$. Now let us denote by T the additional number of steps that pass before these two buckets reach a point where they are about to begin another crossing of each other. Clearly the forward-moving bucket must pass every backward-moving bucket exactly once during these T steps, and the backward-moving bucket must pass each forward-moving bucket exactly once. It is easy to see, then, that $T = P + N/2$ and that after T steps the two buckets are at $v_{f+N/2}$ and $v_{r+N/2}$, respectively. At this point $t = \Delta_j/2 - 1 + P + N/2$, and we claim that all points between v_{s_j+1} and $v_{s_{j+1}}$ have had their tokens broadcast to all locations. This is because each such point has been passed by both of the buckets, and every location on the ring has subsequently been passed by at least one of these two buckets. Therefore, the time taken for this strategy for the entire ring is $\max_{1\leq j\leq P} \Delta_j/2 - 1 + P + N/2$, and we seek to minimize this quantity by choice of P and Δ_j.

Note that if Δ_j and P were real numbers we could find the minimum of the above expression by setting all $\Delta_j = \Delta = \frac{N}{P}$ and using elementary calculus to obtain $\Delta = \sqrt{2N}$, $P = \Delta/2$. This result can be achieved if $M = N/2$ is a square, for then if $a = \sqrt{M}$ we can set $\Delta = 2a$ and $P = a$, obtaining a strategy that completes in $N/2 + \sqrt{2N} - 1$ steps. We come as close to this result as we can by using values Δ_j that are as near as possible to $\sqrt{2N}$. We do this in the following way. First, we must establish the following lemma.

LEMMA 9.1. *Given an integer M and real numbers a, b such that $ab = M$ and $a \leq b$, let $a_L, a_U = a_L + 1, b_L, b_U = b_L + 1$ be integers such that $a_L \leq a \leq a_U$ and $b_L \leq b \leq b_U$; then one can find integers n_L and n_U with $b_L \leq n_L + n_U \leq b_U$ so that:*

$$M = n_L a_L + n_U a_U.$$

Proof. Consider the set

$$S = \left\{ \sum_{b_L \leq \alpha+\beta \leq b_U} \alpha a_L + \beta a_U \right\}.$$

$S = S_1 \cup S_2$ where

$$S_1 = \left\{ \sum_{\alpha+\beta=b_L} \alpha a_L + \beta a_U \right\} \quad \text{and} \quad S_2 = \left\{ \sum_{\alpha+\beta=b_U} \alpha a_L + \beta a_U \right\}.$$

Let

$$m_1 = \min_{s\in S_1} s, \qquad M_1 = \max_{s\in S_1} s, \qquad m_2 = \min_{s\in S_2} s, \qquad M_2 = \max_{s\in S_2} s.$$

Clearly, $m_1 = b_L a_L < m_2 = b_U a_L$ and $M_1 = b_L a_U < M_2 = b_U a_U$.

CLAIM. (1) $m_2 \leq M_1 + 1$;

(2) *S_1 contains every integer between m_1 and M_1; and*

(3) *S_2 contains every integer between m_2 and M_2.*

Proof. (1) $a \le b \Rightarrow a_L \le b_U \Rightarrow a_L b_L + a_L \le a_L b_L + b_L + 1 \Rightarrow a_L(b_L+1) \le (a_L+1)b_L + 1 \Rightarrow b_U a_L \le b_L a_U + 1$;

(2) by induction: S_1 contains m_1, and suppose S_1 contains $x < M_1$; then $x = \alpha a_L + \beta b_U$ where $\alpha > 0$ and $\beta < b_L$ and thus S_1 contains $x+1 = (\alpha-1)a_L + (\beta+1)a_U$;

(3) similar to (2). Now (1), (2), and (3) together imply that S contains every integer between $m_1 = b_L a_L$ and $M_2 = b_U a_U$. Since $m_1 \le M \le M_2$, S must contain M. □

Now using this lemma with $M = N/2, a = b = \sqrt{M}$, we set $P = n_L + n_U$, $\Delta_j = 2a_L$ for $1 \le j \le n_L$, and $\Delta_j = 2a_U$ for $n_L + 1 \le j \le P$. Now observe that $\Delta_j/2 \le a_U \le \sqrt{2N}/2 + 1$ and $P \le b_U \le \sqrt{2N}/2 + 1$, so that

$$\max_{1 \le j \le P} \frac{\Delta_j}{2} - 1 + P + \frac{N}{2} \le \frac{N}{2} + \sqrt{2N} + 1.$$

Now for odd N, we use a slight modification of the same approach and obtain a very similar result. We let $M = (N+1)/2$ and apply the lemma with $a = b = \sqrt{M}$ once more, setting P and all Δ_j as before, except that we set $\Delta_P = 2a_U - 1$. This puts an odd gap between s_P and s_1 and we have to observe the effect. By tracing the steps, it is easy to see that the odd gap stays in front of the backward-moving bucket that starts at s_1 as it travels around the ring. This means that this bucket always experiences bad crossings and it is the only backward-moving bucket to have bad crossings, and also that forward-moving buckets experience bad crossings just when they cross this bucket. For $j \ne P$, we consider v_f and v_b with $f = s_j + \Delta_j/2$ and $b = f + 1$ as before, but this time we find that $T = P + (N+1)/2$ steps occur before the two buckets meet for a second time. In that amount of time the forward moving bucket, which experiences just one bad crossing in passing all backward-moving buckets, advances $(N-1)/2$ links around the ring, while the backward-moving bucket experiences no bad crossings and advances $(N+1)/2$ links; this brings the two buckets to the adjacent sites $v_{f+(N-1)/2}$ and $v_{b+(N-1)/2}$. Now we observe that $\Delta_j/2 \le a_U \le \sqrt{2(N+1)}/2 + 1$ and $P \le b_U \le \sqrt{2(N+1)}/2 + 1$, so that

$$\max_{1 \le j \le P-1} \frac{\Delta_j}{2} - 1 + P + \frac{(N+1)}{2} \le \frac{(N+1)}{2} + \sqrt{2(N+1)} + 1.$$

For $j = P$, we look at v_f and v_b with $f = s_P + (\Delta_P - 1)/2$ and $b = f + 2$. At time $t = (\Delta_P - 1)/2 - 1$ the former is reached by the forward-moving bucket from v_{s_P+1} and the latter is reached by the backward-moving bucket from v_{s_1}. Then the forward-moving bucket experiences just one bad crossing in passing all backward-moving buckets so that in $T = P + (N+1)/2$ more steps it reaches $v_{f+(N+1)/2-1}$. The backward-moving bucket experiences all bad crossings but they do not retard it so it reaches $v_{b+(N+1)/2-1}$ in T steps. These two final sites are two links apart and in one more step the first step of the ensuing bad crossing reaches the intermediate location. This time we observe that $(\Delta_P - 1)/2 \le a_U - 1 \le \sqrt{2(N+1)}/2$ and $P \le b_U \le \sqrt{2(N+1)}/2 + 1$, so that the overall time for $j = P$ is

$$\frac{(\Delta_P - 1)}{2} - 1 + P + \frac{(N+1)}{2} + 1 \le (N+1)2 + \sqrt{2(N+1)} + 1,$$

just as for $j \ne P$.

Now we combine results for even and odd N, expressing the result in a form that will match the lower bound derived in the following section. For even N, the

number of steps is an integral value bounded above by $N/2 + \sqrt{2N} + 1$ and it is therefore bounded by $\lfloor N/2 \rfloor + \lfloor \sqrt{2N} \rfloor + 1 \leq \lfloor N/2 \rfloor + \lceil \sqrt{2N} \rceil + 1$. For odd N, this same reasoning yields a bound of $\lfloor N/2 \rfloor + 1 + \lfloor \sqrt{2N+2} \rfloor + 1 \leq \lfloor N/2 \rfloor + \lceil \sqrt{2N} \rceil + 2$. Combining these gives the following upper bound.

THEOREM 9.2. *The gossip problem for the N-node ring under the* H1 *model of communication can be solved in*

$$\left\lfloor \frac{N}{2} \right\rfloor + \left\lceil \sqrt{2N} \right\rceil + 2$$

steps.

9.2. Lower bound. In establishing a lower bound for the ring, we must treat transmissions that go in different directions separately. We arbitrarily designate one direction around the ring as the forward direction and the other as the backward direction. We suppose that we have a strategy for the ring that completes in T steps, and we shall establish a lower bound for T. Let $s_t^F(v)$ ($s_t^B(v)$) be the forward-most (backward-most) site to which v's token is carried by time T via forward (backward) transmissions in the given strategy. Let $P^B(v)$ ($P^F(v)$) be the backward-most (forward-most) point w for which $s_T^F(w) = s_T^F(v)$ ($s_T^B(w) = s_T^B(v)$). The token from $P^B(v)$ going in the forward direction is, by time T, moving along with the token from v, and it is the backward-most such token. Now for each location v, we form a set $G(v)$ containing all points starting with $P^B(v)$ forward through v, up to and including $P^F(v)$, and we consider the collection $S = \{G(v)\}$. S is partially ordered by set inclusion and we take the set S' of maximal elements under this order. This gives us sets we shall denote as $U_1, U_2, \cdots, U_K$. These sets correspond to the buckets used in the algorithm of the previous section. We have defined these sets according to conditions at the ending time T, and we cannot assume that all tokens are first organized into the buckets and then sent around the ring. Instead we must admit that tokens may coalesce at any step, although we shall in the end show that coalescing at the beginning is optimal.

The buckets as we have defined them here have a number of important properties. Each U_i consists of a contiguous set of vertices of the ring whose backward-most point we shall denote by B_i and forward-most point by F_i. All B_i are distinct, and all F_j are distinct (otherwise one set would be included in another and hence not maximal). In general the buckets can overlap, although in an optimal solution they do not. The maximality of the U_i means that if tokens from some bucket U_j going backward cross tokens from U_{i_2} and then tokens from U_{i_1} before time T, they must cross them at different times: the token from B_{i_1} must be crossed later than the token from B_{i_2} and hence later than any token from U_{i_2}. Similarly, tokens from U_j going forward must cross all tokens from U_{i_1} before crossing the token from F_{i_2}. This means that we can bound the number of crossings by counting the buckets. And finally we observe that the token exchange cannot be complete until for each i backward-moving transmissions have carried the token from F_i and forward-moving transmissions have carried the token from B_i past each other once and then more or less halfway around the ring until they are adjacent and about to pass each other again. To see this, simply observe that by construction, there is some v whose token has been carried no further by backward transmissions than the token from F_i, and no further by forward transmissions than the token from B_i, by time T. Thus if v's token has been broadcast to all points by time T, then the tokens from F_i and B_i have been carried

to the positions described above.

Let $i \neq j$ be given and let t_1 be the latest time such that $\{s_t^F(B_i) \mid 0 \leq t \leq t_1\}$ and $\{s_t^B(F_j) \mid 0 \leq t \leq t_1\}$ have an empty intersection. Let t_2 be the earliest time such that $\{s_t^F(B_i) \mid 0 \leq t \leq t_2\}$ and $\{s_t^B(F_j) \mid 0 \leq t \leq t_2\}$ have an intersection consisting of at least 2 elements. (As long as $i \neq j$ it is not difficult to see that there must be such values t_1 and t_2: the token from F_i stays "behind" the token from F_j in moving in the backward direction, and by $t = T$ meets the token from B_i, so that the token from F_j must have met the token from B_i before that and gone past it.) Before t_1 and after t_2 the tokens from B_i and F_j can move at the same time; between t_1 and t_2 we say that the two tokens have experienced a *crossing*. Define $D_{i,j}^F$ to be $t_2 - t_1 - d^F$ where d^F is the distance between $s_{t_2}^F(B_i)$ and $s_{t_1}^F(B_i)$; similarly define $D_{j,i}^B$ to be $t_2 - t_1 - d^B$ where d^B is the distance between $s_{t_2}^B(F_j)$ and $s_{t_1}^B(F_j)$. $D_{i,j}^F$ is the delay experienced in the given strategy by the forward-moving token from B_i in crossing the backward-moving token from F_j, and $D_{j,i}^F$ is the delay experienced by the backward-moving token from F_j in crossing the forward-moving token from B_i. It is easy to see that $D_{i,j}^F + D_{j,i}^B \geq 2$; this is because the forward and backward transmissions required to cross the tokens cannot occur in parallel, there being at least two steps where one but not the other token is advanced. With only a little extra effort we can now also include the case $i = j$ and obtain the same inequality. If $B_i \neq F_i$, then all the above definitions and arguments go through and we immediately have $D_{i,i}^F + D_{i,i}^B \geq 2$. Otherwise we define $t_1 = 0$ and t_2 as before, so that $D_{i,i}^F$ and $D_{i,i}^B$ can be defined as above with the same interpretation as the crossing delays for the forward-moving token from B_i and the backward-moving token from F_i. Now, conveniently, we still have the inequality $D_{i,i}^F + D_{i,i}^B \geq 2$ (actually $D_{i,i}^F + D_{i,i}^B \geq 3$ is true); this is because the tokens adjacent to $B_i = F_i$ on either side must move first before the token at B_i can (otherwise, U_i would not be maximal). Thus in all cases we have $D_{i,j}^F + D_{j,i}^B \geq 2$.

Now we can write some inequalities based on the above observations. For each i we look at what is required to complete the strategy and what delays are involved. Let x be the distance traveled by the token from F_i in backward transmissions and y the distance traveled by the token from B_i in forward transmissions. By the above analysis, we have

$$T \geq x + \sum_{j=1}^{K} D_{i,j}^B$$

and

$$T \geq y + \sum_{j=1}^{K} D_{i,j}^F.$$

Now we add these two inequalities and observe that since B_i and F_i must meet, we must have $x + y \geq N - 2 + C_i$, where $C_i = |U_i|$.

$$2T \geq N - 2 + C_i + \sum_j D_{i,j}^F + \sum_j D_{i,j}^B.$$

To establish a lower bound for T, we seek the minimum of the above expression. First we sum over i and use the facts that $\sum_i C_i \geq N$ and $D_{i,j}^F + D_{j,i}^B \geq 2$.

$$T \geq \frac{N}{2} + K + \frac{N}{2K} - 1.$$

Then elementary calculus can be used to find the minimal value of T.

$$K = \sqrt{N/2},$$
$$T \geq \frac{N}{2} + \sqrt{2N} - 1.$$

Since the number of steps is an integral quantity, we can express the lower bound in the following way:

THEOREM 9.3. *The gossip problem for the N-node ring under the* H1 *model of communication requires at least*

$$\left\lfloor \frac{N}{2} \right\rfloor + \left\lceil \sqrt{2N} \right\rceil - 1$$

steps.

The algorithm establishing the upper bound that we gave above actually achieves this when $2N$ is a square, so this bound is tight.

10. Other topologies. It was previously shown that the determination of optimal solutions to the gossip problem for general graphs is an NP-complete problem. At the same time, there is a large class of regular graphs that have been proposed as multiprocessor interconnection networks but that not have been discussed in this paper. Such classes include trees, pyramids, cube-connected cycles, and shuffle exchange graphs. Our decision not to expand the present work with a comprehensive treatment of those classes is based on one primary factor: the possibility for using such classes in actual systems seems unlikely at this point in time.

The Fibonacci algorithm of §5.2 only uses $O(\log N)$ edges and those edges define an interconnection network that is ideal for the gossip problem. This network has some 50 percent more edges than the hypercube, a slightly smaller diameter, and it supports a solution to the gossip problem that is about 20 percent faster than our best solution for the hypercube.

We can give one result that complements the lower bound of $1.44 \lg N$ for arbitrary graphs. Given an arbitrary graph, we can choose a spanning tree and use a compression-expansion approach by sending all tokens to the root and then sending the complete collection back. Advancing tokens from one level in the tree to the next level may cost anywhere between 1 step and $K-1$ steps (K steps at the root) where K is the maximum degree of any node, depending on the subtrees, since if subtrees differ in their timing requirements the $K-1$ transmissions can be overlapped. (This is determined as in the well-known labeling algorithm used for compiler code generation [1, p. 541].) For a graph of radius R whose nodes have at most K edges each, this yields a solution in at most $2(K-1)R+2$ steps.

THEOREM 10.1. *The gossip problem for an arbitrary connected graph of radius R and valence K can be solved in $2(K-1)R+2$ steps under the* H1 *model of communication.*

Proof. We choose a spanning tree of height R, send all tokens to the root in at most $(K-1)R+1$ steps, and then reverse the process to distribute the collection to all nodes. To construct the spanning tree, choose a *center* as root, i.e., a node at distance no greater than R from every node in the graph. Then in a breadth-first manner add nodes at distances $1, 2, \cdots, R$, adding just one edge when adding a node so that the constructed graph is always a tree. □

TABLE 1
Summary of results. In some cases assumptions are made regarding minimum sizes of the graphs.

RESULTS FOR THE H1 MODEL OF COMMUNICATION			
GRAPH	DIAMETER	LOWER BOUND	UPPER BOUND
Complete N nodes	1	$\approx 1.44 \lg N$	$\approx 1.44 \lg N$ [8]
Hypercube $N = 2^d$ nodes	$D = \lg N$	$\approx 1.44\, D$	$\approx 1.88\, D$ [18]
Toroidal grid $n_1 \times \cdots \times n_d$	$D = \lfloor \frac{n_1}{2} \rfloor + \cdots + \lfloor \frac{n_d}{2} \rfloor$	D	$D + 18d + 39$
Grid $n_1 \times \cdots \times n_d$	$D = n_1 + \cdots + n_d - d$	D	D
Ring N nodes	$D = \lfloor N/2 \rfloor$	$D + \lceil \sqrt{2N} \rceil - 1$	$D + \lceil \sqrt{2N} \rceil + 2$

11. Conclusions.

11.1. Summary. Table 1 summarizes what we know of the time complexity of the gossip problem under the H1 model of communication. The five types of graphs are listed in order of increasing diameter relative to their size. The diameter always provides a lower bound, and generally graphs with larger diameter have a better chance of having solutions with a number of steps close to the diameter. Solution times of 2 $\lg N$ for an N-node graph are at a borderline where the solution time departs from the diameter.

Solutions can be analyzed in terms of the spanning tree compression and expansion approach of §10. The parallel-transmission methods of §6, if the steps are organized in a certain order and needless transmissions deleted, use compression and expansion along spanning trees and the number of steps used is the best possible for a compression/expansion approach. Furthermore, these trees are optimal in that among all trees supporting distribution of tokens in a given number of steps, they have the most nodes. By contrast, the solutions for complete graphs, regular and toroidal grids, and rings do not use a compression/expansion approach and they are faster than is possible under such an approach. Instead, they can be seen to use a parallel-compression approach where each node in the graph serves as the root of a distinct spanning tree on which a compression-only approach is used. The best known algorithm for the hypercube [18] uses a hybrid method: compression occurs in parallel on a limited number of spanning trees (5 for the 9-cube, 25 for the 17-cube) followed by expansion.

11.2. Open problems.

1. For graphs with low diameters such as the hypercube, we wonder what factors have the strongest influences on the time required to gossip under H1: is it the valence at the nodes, or the diameter, or the radius, or the specific geometry of the interconnect? That is, if all other factors are equal but two graphs differ just in valence, or diameter, or radius, or interconnect pattern, how is the time complexity of the gossip problem likely to differ for them? Comparison of the hypercube with the optimal interconnect pattern defined by the Fibonacci algorithm does not shed light on this since the graphs differ in all regards: the 16-node hypercube has degree 4, radius 4, diameter 4, and a solution in

eight steps while the 16-node optimal graph has degree 5, radius 3, diameter 3, and a solution in six steps; similarly, the 64-node hypercube has degree 6, diameter 6, radius 6, and a solution in 12 steps while the 68-node optimal graph has degree 8, diameter 4, radius 4, and a solution in nine steps.

We also note that it is possible for each node in a hypercube to send its token to its antipode (most distant node) in $D + 1$ steps: two-color the cube red and black, then have red nodes transmit through dimension 1, then black ones through dimension 2, then red ones through dimension 3, and so on, finishing with a transmission through dimension 1. Hence the distance that tokens must travel does not seem to be a dominating factor, although our intuition is that it has some influence on the speed with which the gossip problem can be solved.

2. How can the gap between $1.44D$ and $1.88D$ for the hypercube under H1 be narrowed?
3. How many steps are required to gossip on small two-dimensional grids under H1?
4. What is the time complexity of the gossip problem under Hn and other models? (See [8], [16] for some nonoptimal results.)
5. A graph is *vertex-transitive* if given any nodes v and w there is an isomorphism carrying v to w. (Such graphs are favored for multiprocessor topologies because of mass-production concerns.) We have found no counterexamples to the following conjecture: If G is a vertex-transitive graph with N nodes and diameter (and radius) D, then the gossip problem under the H1 model can be solved in $2 \max(D, \lg N)$ steps. The hypercube comes closest to this bound as far as we know: for the hypercube, $D = \lg N$ and the best solution we know of takes almost $2D$ steps.

Acknowledgment. The referee's thoroughness and helpfulness are gratefully acknowledged.

REFERENCES

[1] A. V. AHO AND J. D. ULLMAN, *Principles of Compiler Design*, First Edition, Addison–Wesley, 1977.

[2] N. ALON, A. BARAK, AND U. MANBER, *On disseminating information reliably without broadcasting*, Tech. Report TR 621, Department of Computer Science, University of Wisconsin, Madison, WI, 1985.

[3] N. S. ARENSTORF AND H. F. JORDAN, *Comparing barrier algorithms*, Tech. Report, ICASE Report 87-65, NASA-Langley Research Center, Hampton, VA, 1987.

[4] A. BAGCHI, S. L. HAKIMI, J. MITCHEM, AND E. SCHMEICHEL, *Parallel algorithms for gossiping by mail*, Inform. Process. Lett., 34 (1990), pp. 197–202.

[5] A. BORODIN AND J. E. HOPCROFT, *Routing, merging, and sorting on parallel models of computation*, J. Comput. System Sci., 30 (1985), pp. 130–145.

[6] S. COOK, C. DWORK, AND R. REISCHUK, *Upper and lower bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 15 (1986), pp. 87–97.

[7] L. CSANKY, *Fast parallel matrix inversion algorithms*, SIAM J. Comput., 5 (1976), pp. 618–623.

[8] R. C. ENTRINGER AND P. J. SLATER, *Gossips and telegraphs*, J. Franklin Inst., 307 (1979), pp. 353–360.

[9] S. EVEN AND B. MONIEN, *On the number of rounds necessary to disseminate information*, in Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures, 1989, Santa Fe, NM, pp. 318–327.

[10] A. M. FARLEY AND A. PROSKUROWSKI, *Gossiping in grid graphs*, J. Combin. Inform. System Sci., 5 (1980), pp. 161–172.

[11] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability. A Guide to the Theory of*

NP-*Completeness*, W.H. Freeman, San Francisco, CA, 1979.

[12] W. M. GENTLEMAN, *Some complexity results for matrix computations on parallel processors*, J. Assoc. Comput. Mach., 25 (1978), pp. 112–115.

[13] G. H. GOLUB AND C. VAN LOAN, *Matrix Computations*, The Johns Hopkins University Press, Baltimore, MD, 1984.

[14] J. L. GUSTAFSON, S. HAWKINSON, AND K. SCOTT, *The architecture of a homogeneous vector supercomputer*, in Proc. 1986 Internat. Conference on Parallel Processing, 1986, St. Charles, IL, pp. 649–652.

[15] M. T. HEATH, ED., *Hypercube Multiprocessors* 1986, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1986.

[16] S. M. HEDETNIEMI, S. T. HEDETNIEMI, AND A. L. LIESTMAN, *A survey of gossiping and broadcasting in communications networks*, Networks, 18 (1988), pp. 320–349.

[17] W. KNODEL, *New gossips and telephones*, Discr. Math., 13 (1975), pp. 95.

[18] D. W. KRUMME, *Fast gossiping for the hypercube*, SIAM J. Comput., 21 (1992), to appear.

[19] D. W. KRUMME, K. N. VENKATARAMAN, AND G. CYBENKO, *The token exchange problem*, Tech. Report 88-2, Department of Computer Science, Tufts University, Medford, MA, 1988.

[20] ———, *Gossiping in minimal time*, Tech. Report 1027, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1990.

[21] V. PAN AND J. REIF, *Fast and efficient solutions of linear systems*, Tech. Report TR-02-85, Center for Research in Computer Technology, Harvard University, Cambridge, MA, 1985.

[22] D. PELEG AND E. UPFAL, *The token distribution problem*, Tech. Report, IBM Almaden Research Center, San Jose, CA, 1986.

[23] Y. SAAD AND M. H. SCHULTZ, *Data communications in parallel architectures*, Tech. Report RR YaleU/DCS/RR-461, Department of Computer Science, Yale University, New Haven, CT, 1986.

[24] P. SCHMITT, *Spreading information by conferences*, Discr. Math., 15 (1976), pp. 305–306.

[25] C. SEITZ, *The cosmic cube*, Comm. ACM, 28 (1985), pp. 22–33.

[26] E. M. STEIN AND G. WEISS, *Introduction to Fourier Analysis on Euclidean Spaces*, Princeton University Press, Princeton, NJ, 1971.

[27] Q. F. STOUT AND B. WAGER, *Intensive hypercube communications:* I, Tech. Report CRL-TR-9-87, Computing Research Laboratory, University of Michigan, Ann Arbor, MI, 1987.

[28] K. N. VENKATARAMAN, G. CYBENKO, AND D. W. KRUMME, *Simultaneous broadcasting in multiprocessor networks*, in Proc. 1986 Internat. Conference on Parallel Processing, 1986, St. Charles IL, pp. 555–578.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 140–150, February 1992
© 1992 Society for Industrial and Applied Mathematics
011

USING INTERIOR-POINT METHODS FOR FAST PARALLEL ALGORITHMS FOR BIPARTITE MATCHING AND RELATED PROBLEMS*

ANDREW V. GOLDBERG†, SERGE A. PLOTKIN†, DAVID B. SHMOYS‡, AND EVA TARDOS‡

Abstract. In this paper interior-point methods for linear programming, developed in the context of sequential computation, are used to obtain a parallel algorithm for the bipartite matching problem. This algorithm finds a maximum cardinality matching in a bipartite graph with n nodes and m edges in $O(\sqrt{m}\log^3 n)$ time on a CRCW PRAM. The results here extend to the weighted bipartite matching problem and to the zero-one minimum-cost flow problem, yielding $O(\sqrt{m}\log^2 n \log nC)$ algorithms, where $C > 1$ is an upper bound on the absolute value of the integral weights or costs in the two problems, respectively. The results here improve previous bounds on these problems and introduce interior-point methods to the context of parallel algorithm design.

Key words. linear programming, interior-point methods, parallel algorithms, matching

AMS(MOS) subject classification. 68Q25

1. Introduction. In this paper we use interior-point methods for linear programming, developed in the context of sequential computation, to obtain a parallel algorithm for the bipartite matching problem. Although Karp, Upfal, and Wigderson [6] have shown that the bipartite matching problem is in RNC (see also [12]), this problem is not known to be in NC. Special cases of the problem are known to be in NC. Lev, Pippenger, and Valiant [9] gave an NC algorithm to find a perfect matching in a regular bipartite graph. Miller and Naor [10] gave an NC algorithm to decide whether a planar bipartite graph has a perfect matching.

The best previously known deterministic algorithm for the problem, due to Goldberg, Plotkin, and Vaidya [4], runs in $O^*(n^{2/3})$ time on graphs with n nodes, where an algorithm runs in $O^*(f(n))$ time if it runs in $O(f(n)\log^k(n))$ time for some constant k. In this paper we describe an $O^*(\sqrt{m})$ algorithm to find a maximum cardinality matching in a bipartite graph with m edges, which is based on an interior-point algorithm for linear programming and on Gabow's algorithm [3] for edge-coloring bipartite graphs. For graphs of low-to-moderate density, this bound is better than the bound mentioned above.

* Received by the editors September 8, 1989; accepted for publication (in revised form) April 3, 1991.

† Department of Computer Science, Stanford University, Stanford, California 94305. The research of the first author was partially supported by National Science Foundation Presidential Young Investigator grant CCR-8858097, with matching funds from AT&T and Digital Equipment Corporation, IBM Faculty Development Award, a grant from 3M Corporation, and Office of Naval Research contract N00014–88–K–0166. The research of the second author was supported by National Science Foundation Research Initiation award CCR900-8226 and by Office of Naval Research contract N00014–88–K–0166.

‡ School of Operations Research and Industrial Engineering, Cornell University, Ithaca, New York 14853. The research of the third author was partially supported by a National Science Foundation Presidential Young Investigator award CCR-89-96272 with matching support from United Parcel Service, and Sun Microsystems, by Air Force contract AFOSR-86–0078, and by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through National Science Foundation grant DMS-8920550, as well as the National Science Foundation Presidential Young Investigator award of the first author. The research of the fourth author was supported in part by a Packard Research Fellowship and by the National Science Foundation, the Air Force Office of Scientific Research, and the Office of Naval Research, through National Science Foundation grant DMS-8920550 and by Air Force contract AFOSR-86–0078, as well as by the National Science Foundation Presidential Young Investigator award of the first author.

The results presented in this paper extend to the maximum-weight matching problem and to the zero-one minimum-cost flow problem. The resulting algorithms run in $O^*(\sqrt{m}\log C)$ time, where $C > 1$ is an upper bound on the absolute value of the integral weights and costs in the two problems, respectively. The best previously known algorithm for the zero-one minimum-cost flow problem runs in $O^*((nm)^{2/5}\log C)$ time [4]. The new algorithm is better for both the zero-one maximum flow and the zero-one minimum-cost flow problems for all graph densities.

Interior-point algorithms work as follows. The algorithm starts with a point in the interior of the feasible region of the linear program and its dual that is close to the so-called central path. In its main loop, the algorithm moves from one interior point to another, decreasing the value of the duality gap at each iteration. When this value is small enough, the algorithm terminates with an interior-point solution that has a near-optimal value. The finish-up stage of the algorithm converts this near-optimal solution into an optimal basic solution.

Karmarkar's revolutionary paper [5] spurred the development of the area of interior-point linear programming algorithms, and many papers have followed his lead. Karmarkar's algorithm runs in $O(NL)$ iterations, where N and L denote the number of variables and the size of the linear program, respectively. Renegar [15] was the first to give an interior-point algorithm that runs in $O(\sqrt{N}L)$ iterations. Since then, several different $O(\sqrt{N}L)$-iteration algorithms have been developed. For an overview of work on interior-point algorithms, the reader is referred to the survey paper of Todd [17]. The matching algorithm discussed in this paper is based on an algorithm due to Monteiro and Adler [11], though similar algorithms can also be based on other $O(\sqrt{N}L)$ iteration algorithms.

Interior-point algorithms have proved to be an important tool for developing efficient sequential algorithms for linear programming and its special cases. In this paper we apply these tools in the context of parallel computation. For the purpose of parallel computation, an important fact is that the running time of an iteration of an interior-point algorithm is dominated by the time required for matrix multiplication and inversion. Therefore, an iteration of such an algorithm can be done in $O(\log^2 N)$ time on a CRCW PRAM using N^3 processors [13].

The interior-point method used here follows a central path in the interior of the feasible region. After every $\sqrt{N}$ iterations, this algorithm has decreased the duality gap by a constant factor. The bipartite matching problem can be formulated as a linear program with an integral optimum value. Therefore the size of the maximum matching is known as soon as this gap is below one. Furthermore, the gap between the value of an initial solution and the optimal value is at most N. This suggests that an interior-point algorithm can be used to find the value of the maximum matching in a bipartite graph in $O(\sqrt{m}\log n)$ iterations, or $O^*(\sqrt{m})$ time. In this paper we develop an algorithm running in this time bound that finds a maximum matching as well as its value.

To find a maximum matching we need to overcome two difficulties. First, we need to find an initial interior point and dual solution that is close to the central path and has a small duality gap, so that the number of iterations will be small. The second difficulty comes from the fact that standard implementations of the finish-up stage of interior-point algorithms either are inherently sequential or perturb the input problem to simplify the finish-up stage, increasing the number of iterations of the main loop by an $\Omega(n)$ factor. For the special case of the bipartite matching problem, we give a parallel implementation of the finish-up stage that runs in $O(\log^2 n)$ time using m

processors. This implementation is based on Gabow's edge-coloring algorithm [3].

Our techniques apply to the more general maximum-weight matching problem. The algorithm and its analysis are only slightly more involved. For brevity we focus on the more general case. The results for the maximum matching problem are obtained as a simple corollary of the results for the maximum-weight matching problem. The main loop of our maximum-weight matching algorithm runs in $O(\sqrt{m}\log^2 n \log nC)$ time and uses m^3 processors, and the finish-up stage runs in $O(\log n \log nC)$ time and uses m processors. Therefore, the algorithm runs in $O^*(\sqrt{m}\log C)$ time. A standard reduction between the weighted matching and the zero-one minimum-cost flow problems (see, e.g., [1], [6]) gives an $O^*(\sqrt{m}\log C)$ algorithm.

This paper is organized as follows. Section 2 introduces definitions and terminology used throughout the paper and reviews the Monteiro–Adler linear programming algorithm. Section 3 gives a linear programming formulation of the bipartite matching problem that has an initial interior point close to the central path with a small duality gap, and shows how to use the linear programming algorithm to obtain a near-optimal fractional matching. Section 4 describes a parallel procedure that, in $O^*(\log C)$ time, converts the near-optimal fractional matching into an optimal zero-one matching. Section 5 contains concluding remarks.

2. Preliminaries. In this section we define the matching problem and the linear programming problem, and review some fundamental facts about them. For a detailed treatment, the reader is referred to the textbooks by Papadimitriou and Steiglitz [14] or Schrijver [16]. We also give an overview of the Monteiro–Adler algorithm.

The *bipartite matching problem* is to find a maximum cardinality matching in a bipartite graph $G = (V, E)$. The *maximum-weight bipartite matching problem* is defined by a bipartite graph $G = (V, E)$ and a weight function on the edges $w : E \longrightarrow \mathbf{R}$. We shall assume that the weights are integral. The *weight* of a matching M is $\sum_{e\in M} w(e)$. The problem is to find a matching with maximum weight.

We use the following notation and assumptions. Let $G = (V, E)$ denote the (bipartite) input graph, let n denote the number of nodes in G, let m denote the number of edges in G, and let C denote the maximum absolute value of the weights of edges in G. To simplify the running time bounds, we assume, without loss of generality, that $m \geq n-1 > 1$, and $C > 1$. We denote the degree of a node v by $d(v)$, and the set of edges incident to node v by $\delta(v)$. For a vector x, we let $x(i)$ denote the ith coordinate of x. We use a PRAM [2] as our model of parallel computation.

Consider the following standard linear programming formulation of the bipartite matching problem.

$$\textit{Matching-1:} \quad \begin{array}{lrll} \text{maximize} & w^t f & & \\ \text{subject to:} & \sum_{e\in\delta(v)} f(e) & \leq 1, & \text{for } v = 1, \cdots, n, \\ & f & \geq 0. & \end{array}$$

A feasible solution to the system of the linear inequalities above is called a *fractional matching.* We denote an optimal solution of the linear program by f^*.

The constraint matrix of *Matching*-1 is the node-edge incidence matrix of the bipartite graph G. A matrix is *totally unimodular* if all of its submatrices have determinants $+1$, -1, or zero. It is well known that the node-edge incidence matrix of a bipartite graph is totally unimodular [14]. This implies the following theorem.

THEOREM 2.1. [14] *Any optimal solution of the linear program Matching-1 is the convex combination of maximum-weight matchings. The optimal value of this linear*

program is equal to the maximum weight of a matching.

The Monteiro–Adler algorithm handles linear programs in the following form:

$$\text{\textit{Primal} LP:} \quad \begin{array}{ll} \text{minimize} & c^t x \\ \text{subject to:} & Ax = b, \\ & x \geq 0 \end{array}$$

where A is a matrix, and b, c, and x are vectors of the appropriate dimensions. We assume that the matrix A and the vectors b and c are integral. We use N to denote the number of variables in the (primal) linear programs we consider. A vector x is a *feasible solution* if it satisfies the constraints $Ax = b$ and $x \geq 0$. A feasible solution x is *optimal* if it minimizes the objective function value $c^t x$, and is an *interior point* if it is in the interior of the nonnegative orthant.

The linear programming duality theorem states that the minimum value of the Primal LP is equal to the maximum value of the following *Dual* LP:

$$\text{\textit{Dual} LP:} \quad \begin{array}{ll} \text{maximize} & b^t \pi \\ \text{subject to:} & A^t \pi + s = c, \\ & s \geq 0 \end{array}$$

where π and s are the variables of the Dual LP, the dimension of π is equal to the dimension of b, and the dimension of s is equal to the dimension of x. Feasible and optimal solutions and interior points for the dual problem are defined in the same way as for the primal problem.

Let x be a feasible solution to the Primal LP, and let (π, s) be a feasible solution to the Dual LP. The value $c^t x$ is an upper bound and $b^t \pi$ is a lower bound on the common optimal value of the two problems. Hence the difference $c^t x - b^t \pi = s^t x$ measures how far the current solutions are from being optimal. This quantity is called the *duality gap*. The *central path* of this pair of linear programs is defined as the set of points with $s(i)x(i)$ identical for every i, $i = 1, \cdots, N$. Note that the complementary slackness conditions state that the pair of primal and dual solutions is optimal if these products are all zero.

The Monteiro–Adler algorithm is applied to a pair of primal and dual linear programs in the above form. The algorithm starts with a vector (x_0, π_0, s_0), where x_0 and (π_0, s_0) are interior points of the primal and dual linear problems, which are in some sense close to the central path. At each iteration of the main loop, the algorithm moves from the current interior point to another interior point, so that the duality gap is decreased by a factor of $(1 - \Omega(1/\sqrt{N}))$ every iteration.

The measure of closeness to the central path required by the algorithm is defined as follows. Consider a primal-dual solution pair (x, π, s). Define $\mu = s^t x / N$, and define the vector σ such that $\sigma(i) = s(i)x(i)$ for $i = 1, \cdots, N$. The solution pair is *close to the central path* if $||\sigma - \mu \mathbf{1}|| \leq \theta \mu$, where $\mathbf{1}$ denotes the vector with all coordinates 1, $|| \cdot ||$ denotes the Euclidean norm, and θ is 0.35, as suggested by Monteiro and Adler.

Monteiro and Adler prove the following theorem.

THEOREM 2.2. [11] *If we have an initial solution (x_0, π_0, s_0) that is close to the central path, then for any constant $\delta > 0$, after $O(\sqrt{N} \log(s_0^t x_0))$ iterations the duality gap $s^t x$ of the current solution (x, π, s) is at most δ.*

To get the algorithm started, one has to provide an initial solution (x_0, π_0, s_0) that is close to the central path. Monteiro and Adler present a way to obtain an equivalent linear programming formulation with such an initial solution. In the next section we give a slightly simplified version of this construction for the bipartite matching problem, for which the initial solution also has a sufficiently small duality gap.

3. Finding a near-optimal solution. In this section we show how to convert the *Matching*-1 linear program into a linear program that is in the form required by the Monteiro–Adler algorithm and has an initial solution close to the central path with a small duality gap. Then we show how to compute a near-optimal fractional matching from the initial solution to this linear program.

We restate the matching problem as follows:

$$\begin{array}{llll} \textit{Matching-2}: & \text{minimize} & -w^t f + \frac{N^2 C}{n-1} z & \\ & \text{subject to:} & \sum_{e\in\delta(v)} f(e) + (n - d(v))g(v) - z = 1, & \text{for } v = 1, \cdots, n, \\ & & \mathbf{1}^t f + \mathbf{1}^t g + y = n + m + 1, & (*) \\ & & f, g, z, y \geq 0. & \end{array}$$

We denote the objective function of this linear program by the vector c, and coefficients of the left-hand side of the constraint $(*)$ by the vector a. The number of variables in this linear program is $m + n + 2 = N$. We denote a feasible solution to *Matching*-2 by $x = (f, g, y, z)$, and a feasible solution of the corresponding dual problem by π and s, where $\pi(i)$, for $i = 1, \cdots, n$, is the dual variable corresponding to the primal constraint for node i, and $\pi(n+1)$ is the dual variable corresponding to the constraint $(*)$.

Intuitively, the transformation works as follows. Variables $g(v)$ are the slack variables introduced to replace inequality constraints by equality constraints. The positive multipliers $(n - d(v))$ scale the slack variables so that there is a feasible solution with all original and slack variables equal. The coefficient of z in the objective function is large enough to guarantee that $z = 0$ in an optimal solution. The variable z is introduced to make it possible to have a starting primal solution with coordinates of f, g, and y equal (for example, to 1). The constraint $(*)$ does not affect the primal problem since y is not in the objective function and, as we have just mentioned, in an optimal solution $z = 0$ and therefore $g^t\mathbf{1} + f^t\mathbf{1} \leq n$ is automatically satisfied. This constraint, however, allows us to obtain an initial solution for the dual problem such that the dual slack variables corresponding to the primal variables f, g, and y are roughly equal. This will imply that the starting solution is close to the central path.

LEMMA 3.1. *If (f, g, y, z) is an optimal solution of Matching-2, then f is an optimal solution to Matching-1.*

Proof. Every solution to *Matching*-1 can be extended to a solution to *Matching*-2 with $z = 0$; this follows from the fact that both f and the slacks in *Matching*-1 are at most 1.

Next we have to show that every optimal solution to *Matching*-2 has $z = 0$. Consider a feasible point $x_1 = (f_1, g_1, z_1, y_1)$ with $z_1 \neq 0$. Since f_1 satisfies $\sum_{e\in\delta(v)} f_1(e) \leq 1 + z_1$ for every node v, decreasing f_1 on some edges, by a total of at most $z_1 n$, converts f_1 into a vector f_2 that is a fractional matching. Above we observed that any fractional matching can be extended to a feasible solution of *Matching*-2. Let x_2 denote a feasible solution extending f_2. If we replace x_1 by x_2, the decrease in the objective function value caused by the reduction in z is $z_1(N^2C/(n-1)) > z_1 NC$. The increase due to the change in f is bounded by $z_1 nC < z_1 NC$. Therefore, the value $c^t x_2$ is smaller than $c^t x_1$, which implies that any optimal solution must have $z = 0$. □

We define initial primal and dual solutions as suggested by the above discussion. The initial primal solution x_0 is defined by

$$f = \mathbf{1}, g = \mathbf{1}, y = 1, z = n - 1.$$

The initial dual solution (π_0, s_0) is defined by

$$\begin{aligned} \pi(i) &= 0, && \text{for } 1 \leq i \leq n, \\ \pi(n+1) &= -N^2C, \\ s(i) &= c(i) + N^2Ca(i) && \text{for } 1 \leq i \leq N. \end{aligned}$$

LEMMA 3.2. *The vectors x_0 and (π_0, s_0) are interior-point solutions of the primal and the dual problems that are close to the central path, and the value of the duality gap is $O(N^3C)$.*

Proof. It is easy to verify that x_0 is a primal solution and (π_0, s_0) is a dual solution. Recall that $N = n + m + 2$. The duality gap is

$$s_0{}^t x_0 = nN^2C + mN^2C - w^t\mathbf{1} + 2N^2C = N^3C - w^t\mathbf{1} = O(N^3C),$$

as required.

Next we have to verify that the initial solution is close to the central path. Let $\mu = s_0^t x_0 / N = N^2C - (w^t\mathbf{1}/N)$, and define the vector σ with coordinates $\sigma(i) = s(i)x(i)$. Consider $\sigma(i) - \mu$ for each type of variable separately. For variables $s(i)$ and $x(i)$ corresponding to z, y, and g, we get

$$|\sigma(i) - \mu| = |s_0(i)x_0(i) - \mu| = |w^t\mathbf{1}/N|.$$

For variables $s(i)$ and $x(i)$ corresponding to f, we get

$$|\sigma(i) - \mu| = |s_0(i)x_0(i) - \mu| = |w^t\mathbf{1}/N - w(i)|.$$

Using these values we get that

$$||\sigma(i) - \mu||^2 \leq N(w^t\mathbf{1}/N)^2 + \sum_i w^2(i) \leq 2NC^2.$$

Since $N \geq 3$, we have that $2NC^2 \leq \theta^2(N^4C^2 - 2NCw^t\mathbf{1}) \leq (\theta\mu)^2$. This proves the lemma. □

Now we are ready to give the $O^*(\sqrt{m}\log C)$-time algorithm to compute the weight of an optimal matching and to find a near-optimal fractional matching. In the next section we show how to convert such a near-optimal fractional matching into an optimal matching.

LEMMA 3.3. *A fractional bipartite matching with weight at most $\frac{1}{2}$ less than the weight of an optimal matching can be computed in $O^*(\sqrt{m}\log C)$ time on a* PRAM *with m^3 processors.*

Proof. By applying Lemma 3.2 and Theorem 2.2 (with $\delta = \frac{1}{4}$), we see that after $O(\sqrt{N}\log(NC)) = O(\sqrt{m}\log(nC))$ iterations of the LP algorithm, we have obtained a point (x, π, s) with a duality gap $x^t s \leq \frac{1}{4}$. Hence we have

$$-w^t f + \frac{N^2C}{n-1}z + w^t f^* \leq \frac{1}{4}, \tag{1}$$

where f^* is an optimal solution to *Matching*-1. Since $z \geq 0$, this implies that $w^t f^* - w^t f \leq \frac{1}{4}$. As in Lemma 3.1, we can argue that f can be converted to a feasible solution of the *Matching*-1 problem by decreasing its value on some of the edges by a total of at most zn. Therefore, $w^t f^* \geq w^t f - znC$. From (1), this implies that $zN^2C/(n-1) \leq \frac{1}{4} + znC$. Thus,

$$z \leq \frac{n-1}{4C(N^2 - n^2 + n)} < \frac{1}{4mC}.$$

Now round all values of f and g down to have a common denominator $4mC$, and denote the rounded solution by f_1, g_1. Clearly, $w^t f^* - w^t f_1 \leq \frac{1}{4} + (mC)/(4mC) \leq \frac{1}{2}$. After the rounding, we have:

$$\sum_{e \in \delta(v)} f_1(e) + (n - d(v))g_1(v) \quad \leq \quad 1 + z.$$

The left-hand side is an integer multiple of $(4mC)^{-1}$ and $z < (4mC)^{-1}$. This implies that

$$\sum_{e \in \delta(v)} f_1(e) + (n - d(v))g_1(v) \leq 1.$$

Hence, the resulting vector f_1 is a fractional matching whose weight is within $\frac{1}{2}$ of the optimum. □

COROLLARY 3.4. *The cardinality of the maximum matching in a bipartite graph can be computed in $O^*(\sqrt{m})$ time using m^3 processors.*

4. The finish-up stage. In the previous section we have shown how to compute, in $O^*(\sqrt{m}\log C)$ time, a fractional bipartite matching with weight at most $\frac{1}{2}$ less than the optimum. In this section we give an $O^*(\log C)$ algorithm for converting any such fractional matching into a maximum-weight matching. Note that for the unweighted bipartite matching, this algorithm runs in polylogarithmic time.

Let f be a fractional bipartite matching that has weight at most $\frac{1}{2}$ less than the maximum weight, and let f^* denote a maximum-weight matching. First we construct a fractional matching f', such that the values of f' have a relatively small common denominator that is a power of two and the weight of f' differs from the maximum weight by less than 1. Define Δ by

$$\Delta = 2^{\lceil \log mC \rceil + 1}.$$

By definition, Δ is a power of 2 and $\Delta = O(mC)$. Let f' be the fractional matching obtained by rounding f down to the nearest multiple of $1/\Delta$. Note that

$$|w^t f - w^t f'| < \frac{mC}{\Delta} = \frac{mC}{2^{\lceil \log mC \rceil + 1}} < \frac{1}{2}.$$

Therefore, $w^t f^* - w^t f' < 1$.

Consider a multigraph $G' = (V, E')$ with the edge set containing $\Delta \cdot f'(e)$ copies of e for each $e \in E$, and no other edges.

LEMMA 4.1. *For any coloring of the edges of G' with Δ colors, there exists a color class that is a maximum-weight matching of G.*

Proof. The proof is by a simple counting argument. The sum of the weights of the color classes is equal to $\Delta w^t f' > \Delta(w^t f^* - 1)$. Since there are Δ color classes, at least one of them has weight above $w^t f^* - 1$. The claim follows from the integrality of w. □

The above lemma implies that, in order to find a maximum-weight matching, it is sufficient to edge-color G' using Δ colors. Since G' is bipartite graph and its maximum degree is bounded by Δ, which is a power of 2, we can use a parallel implementation of Gabow's algorithm [3] to edge-color G' using Δ colors. However, G' has $O(mC)$ edges and therefore the algorithm uses $\Omega(mC)$ processors. In order to reduce the processor requirement, we use a somewhat different algorithm. The algorithm does not use an

```
procedure Round(E, f);
    Δ ← 2^(⌈log mC⌉+1) ;
    f' ← f rounded down to a common denominator of Δ;
    d' ← Δ;
    while d' > 1 do begin
        E0 ← {e | e ∈ E, d' · f'(e) is odd};
        (E1, E2) ← Degree-Split(V, E0);
        W1 ← w(E1);
        W2 ← w(E2);
        if W1 ≥ W2
          then begin
            for e ∈ E1 do f'(e) ← f'(e) + 1/d';
            for e ∈ E2 do f'(e) ← f'(e) − 1/d';
            end;
          else begin
            for e ∈ E2 do f'(e) ← f'(e) + 1/d';
            for e ∈ E1 do f'(e) ← f'(e) − 1/d';
            end;
        d' ← d'/2;
    end;
    return ({e | f'(e) = 1})
end.
```

FIG. 1. *Rounding an approximate fractional matching to an optimal integral one.*

explicit representation of the multigraph, but rather uses a weighted representation of a simple graph. A divide-and-conquer approach is then used to split the (implicit) multigraph so that the bound on the maximum degree of a note is halved, and then recurses on the part with greater weight. A subroutine for finding such a partitioning is also the basis of Gabow's edge-coloring algorithm.

Figure 1 describes the algorithm to find a maximum-weight matching given a near-optimal fractional matching. The algorithm starts by rounding the fractional matching to a small common denominator, as described above. A fractional matching f' with common denominator Δ, can be written as $f' = \frac{1}{2}(f_1 + f_2)$ such that both f_1 and f_2 are fractional matchings with common denominator $\Delta/2$. On edges with $\Delta f'(e)$ even, we can set $f_1(e) = f_2(e) = f'(e)$. Otherwise we set $f_1(e) = f'(e) + 1/\Delta$ and $f_2(e) = f'(e) - 1/\Delta$ or the other way around. Whether to add or to subtract $1/\Delta$ on these edges is decided with the help of the procedure *Degree-Split*. This procedure partitions the edges of a bipartite graph $G_0 = (V, E_0)$ into two classes E_1 and E_2, so that for every node v, the degree of v in the two induced subgraphs differs by at most one. The procedure is used for the graph on V with edges $E_0 = \{e \in E : \Delta f'(e) \text{ is odd}\}$. To obtain f_1 we increase f' on one color class and decrease it on the other one. Both f_1 and f_2 are fractional matchings with common denominator $\Delta/2$. Now f' is replaced by f_1 or f_2 depending on which one has larger weight. This process is iterated $O(\log(mC))$ times, until the current fractional matching is integral. The resulting matching has an integral weight that is more than $w^t f^* - 1$, and therefore the matching is optimal.

The heart of the algorithm is the procedure *Degree-Split* described in Fig. 2. This procedure decomposes the graph into cycles and paths, such that at most one path ends at each node. This can be accomplished by pairing up the edges incident to

procedure *Degree-Split*(V, E);
  Construct a new node set V' by replacing each node $v \in V$ by an independent set of size $\lceil d(v)/2 \rceil$;
  For each node in V, assign its incident edges to nodes in V', so that each node v in V' has $d(v) \leq 2$;
  Edge-color the resulting graph using two colors;
  Return the edges of each color class;
end.

FIG. 2. *Splitting the maximum degree of the graph.*

each node separately. Then we two-color the paths and cycles separately. This gives a two-coloring of the graph where the difference in the degree of a node in the two subgraphs is at most 1.

LEMMA 4.2. *The algorithm Round produces a maximum-weight matching.*

Proof. Consider the parameter d' used in the algorithm in Fig. 1. Initially $d' = \Delta$. Note that after iteration i we have $d' = \Delta/2^i$. We show by induction that after iteration i:

- f' is a fractional matching,
- $w^t f' > w^t f^* - 1$,
- coordinates of f' have common denominator d'.

Initially all three conditions are satisfied. Assuming that all three conditions are satisfied after iteration $i-1$, we prove that they remain satisfied after iteration i. Let d_1 and f_1 denote d' and f' before iteration i and let d_2 and f_2 denote d' and f' after iteration i.

The last claim follows from the fact that the coordinates of f_1 that are odd multiples of $1/d_1$ are adjusted by plus or minus $1/d_1$ in this iteration, and so all coordinates of f_2 are even multiples of $1/d_1$, and hence multiples of $1/d_2$.

The second claim follows from the fact that the components of f_2 that have been increased correspond to edges of greater total weight than those that have been decreased.

Now consider the first claim. By the inductive assumption, $\sum_{e \in \delta(v)} f_1(e) \leq 1$. By the definition of procedure *Degree-Split*, $\sum_{e \in \delta(v)} f_2(e) \leq \sum_{e \in \delta(v)} f_1(e) + 1/d_1 \leq 1 + 1/d_1$. However, we have already seen that f_2 has a common denominator of d_2. Hence, $\sum_{e \in \delta(v)} f_2(e)$ is an integer multiple of $1/d_2 = 2/d_1$ and is therefore at most one.

After $\log \Delta$ iterations we construct an f' that is integral and whose weight is above $w^t f^* - 1$. By the integrality of w, the set of edges where this f' is 1 is the desired maximum-weight matching of the input graph. ◻

LEMMA 4.3. *The procedure Degree-Split partitions the input graph into two graphs with disjoint edge-sets, such that the degrees of any node v in the two graphs differ by at most one. The procedure can be implemented in $O(\log n)$ time.*

Proof. Observe that the graph constructed on V' is bipartite, and the degree of a node is at most two. Therefore the graph consists of paths and even cycles. Hence it can be two-edge-colored in $O(\log m)$ time using m processors [7], [8]. The claim of the lemma follows from the fact that each node $v \in V$ is an end point of at most one path. ◻

LEMMA 4.4. *The algorithm Round runs in $O(\log n \log nC)$ time using m processors.*

Proof. The number of iterations of the loop of the algorithm is $O(\log \Delta) = O(\log nC)$, because d is halved at each iteration. The running time of each iteration is dominated by *Degree-Split*, which takes $O(\log n)$ time by Lemma 4.3. □

THEOREM 4.5. *A maximum-weight matching in a bipartite graph can be computed in $O^*(\sqrt{m}\log C)$ time using m^3 processors.*
The exact running time of our algorithm on a CRCW PRAM is $O(\sqrt{m}\log^2 n \log nC)$, which is the time required to approximately solve the linear program.

COROLLARY 4.6. *A maximum cardinality bipartite matching can be computed in $O^*(\sqrt{m})$ time using m^3 processors.*

5. Concluding remarks. Interior-point methods have proved to be very powerful in the context of sequential computation. In this paper we have shown an application of these methods to the design of parallel algorithms. We believe that these methods will find more applications in the context of parallel computation. We would like to mention the following two research directions.

One direction is to attempt to generalize our result to general linear programming, showing that any linear programming problem can be solved in $O^*(\sqrt{N}L)$ time. This would require a parallel implementation of the finish-up stage of the algorithm that runs in $O^*(\sqrt{N}L)$ time. A related question is whether the problem of finding a vertex of a polytope with the objective function value smaller than that of a given interior point of the polytope is P-complete.

The other direction of research is to attempt to use the special structure of the bipartite matching problem to obtain an interior-point algorithm for this problem that finds an almost-optimal fractional solution in less than $O^*(\sqrt{m})$ time; an $O^*(1)$ bound would be especially interesting, since in combination with results of §4 it would imply that bipartite matching is in NC.

Acknowledgments. We would like to thank an anonymous referee for very useful comments on an earlier version of this paper.

REFERENCES

[1] A. K. CHANDRA, L. STOCKMEYER, AND U. VISHKIN, *Constant depth reducibility*, SIAM J. Comput., 13 (1984), pp. 423–439.
[2] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, in Proc. 10th ACM Symposium on Theory of Computing, 1978, pp. 114–118.
[3] H. N. GABOW, *Using Euler partitions to edge-color bipartite multi-graphs*, Internat. J. Comput. Inform. Sci., 5 (1976), pp. 345–355.
[4] A. V. GOLDBERG, S. A. PLOTKIN, AND P. M. VAIDYA, *Sublinear-time parallel algorithms for matching and related problems*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, 1988, pp. 174–185.
[5] N. KARMARKAR, *A new polynomial-time algorithm for linear programming*, Combinatorica, 4 (1984), pp. 373–395.
[6] R. M. KARP, E. UPFAL, AND A. WIGDERSON, *Constructing a maximum matching is in random* NC, Combinatorica, 6 (1986), pp. 35–48.
[7] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computation*, J. Assoc. Comput. Mach., 27 (1980), pp. 831–838.
[8] C. LEISERSON AND B. MAGGS, *Communication-efficient parallel graph algorithms*, in Proc. Internat. Conference on Parallel Processing, 1986, pp. 861–868.
[9] G. F. LEV, N. PIPPENGER, AND L. G. VALIANT, *A fast parallel algorithm for routing in permutation networks*, IEEE Trans. Comput., C-30 (1981), pp. 93–100.
[10] G. L. MILLER AND J. NAOR, *Flow in planar graphs with multiple sources and sinks*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 112–117.
[11] R. D. C. MONTEIRO AND I. ADLER, *Interior path following primal-dual algorithms. Part* I: *Linear programming*, Math. Programming, 44 (1989), pp. 27–41.

[12] K. MULMULEY, U. V. VAZIRANI, AND V. V. VAZIRANI, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–113.

[13] V. PAN AND J. REIF, *Efficient parallel solution of linear systems*, in Proc. 17th ACM Symposium on Theory of Computing, 1985, pp. 143–152.

[14] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice–Hall, Englewood Cliffs, NJ, 1982.

[15] J. RENEGAR, *A polynomial-time algorithm, based on Newton's method, for linear programming*, Math. Programming, 40 (1988), pp. 59–93.

[16] A. SCHRIJVER, *Theory of Linear and Integer Programming*, John Wiley & Sons, New York, 1986.

[17] M. J. TODD, *Recent developments and new directions in linear programming*, in Mathematical Programming: Recent Developments and Applications, Kluwer Academic Publishers, Boston, 1989, pp. 109–159.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 151–162, February 1992

© 1992 Society for Industrial and Applied Mathematics
012

EFFICIENT EMBEDDINGS OF TREES IN HYPERCUBES*

SANDEEP N. BHATT†, FAN R. K. CHUNG‡, F. THOMSON LEIGHTON§, AND ARNOLD L. ROSENBERG¶

Abstract. The boolean hypercube is a particularly versatile network for parallel computing. It is well known that multidimensional grid machines can be simulated on a hypercube with no communications overhead. In this paper it is shown that every bounded-degree tree can be simulated on the hypercube with constant communications overhead. In fact, the proof shows that every bounded-degree graph with an $O(1)$-separator can be embedded in a hypercube of the same size with dilation and congestion both $O(1)$. It is also proved that not all bounded-degree graphs can be efficiently embedded within the hypercube.

Key words. graph embedding, binary trees, boolean hypercube, dilation, expansion, congestion, tree decomposition

AMS(MOS) subject classifications. 68M10, 68Q10, 68R05, 68R10

1. Introduction. The binary hypercube is emerging as one of the most popular network architectures for parallel machines. This is due partly to the facts that the hypercube has a simple recursive structure and that there are simple algorithms for message routing on the hypercube that work well in practice.

Another important consideration in the choice of network architecture is its ability to accommodate different algorithms efficiently. The problem of efficiently implementing various algorithms on parallel architectures has traditionally been studied as the "logical mapping problem" [2], [10] so that the problem of implementation becomes one of embedding the "data-dependency graph" underlying an algorithm within the processor interconnection graph. Many structured algorithms such as those in linear algebra [20] or the FFT algorithm [17] can be efficiently mapped onto the hypercube with minimal communication overhead. As an example, the N-node hypercube contains every N-node multidimensional grid, each of whose sides is a power of 2, as a subgraph. Hence grid-based algorithms can be executed efficiently on hypercubes.

In this paper we examine the ability of the hypercube to accommodate divide-and-conquer algorithms whose underlying structures are bounded-degree trees. Our main result is that the N-node hypercube can emulate every N-node bounded-degree tree with only a constant factor slowdown. In particular, we show how to embed any N-node bounded-degree tree within an N-node hypercube so that:

* Received by the editors December 19, 1990; accepted for publication January 28, 1991. This research was supported in part by Bell Communications Research.

† Department of Computer Science, Yale University, New Haven, Connecticut 06520. This author's research was supported by National Science Foundation grants MIPS-86-01885 and CCR-88-07426, by Defense Advanced Research Projects Agency grant CCR-89-08285, and by Air Force grant AFOSR-89-0382.

‡ Bell Communications Research, Morristown, New Jersey 07960.

§ Department of Mathematics and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. This author's research was supported by Air Force Office of Scientific Research grants AFOSR-86-0078 and AFOSR-89-0271, Defense Advanced Research Projects Agency grants N00014-80-C-0622 and N00014-89-J-1988, Army grant DAAL-03-86-K-0171, and a National Science Foundation Presidential Young Investigator Award with matching funds from Xerox and IBM.

¶ Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003. This author's research was supported by National Science Foundation grants DCI-85-04308, DCI-87-96236, and CCR-88-12567.

1. The mapping of tree nodes to hypercube nodes is one-to-one (i.e., the *max-load* of the embedding is one).
2. Each tree edge is mapped onto a hypercube path of length $O(1)$ (i.e., the *dilation* of the embedding is constant). The dilation of an embedding is a lower bound on communication delay, measured by the number of links a message must traverse.
3. Each hypercube edge is used to route only $O(1)$ tree edges (i.e., the *congestion* of the embedding is constant). Congestion bounds message-throughput rates, and thereby communication delay and queue sizes for holding messages in transit.
4. Only $O(1)$ tree edges are routed through each hypercube node (i.e., the *node-congestion* of the embedding is constant). Node-congestion bounds the total queue-size required at each node to hold messages in transit.

In other words, every tree can be embedded into a hypercube with expansion 1, and every other resource bounded by a constant. The embedding uses a divide-and-conquer approach involving multicolor separator theorems for binary trees, and is reminiscent of earlier work [8] on embedding graphs in grids for VLSI layout. Our techniques in this paper consequently translate into efficient embeddings (all resources bounded by a constant) for all bounded-degree graphs with $O(1)$-separators. Bounded-degree trees fall within this class, as do bounded-degree outerplanar graphs.

The paper is organized as follows. Section 2 summarizes related results; §3 presents a simple lower bound for embedding random trivalent graphs in hypercubes; §4 presents the basic technique for decomposing binary trees; §5 presents the final embedding. Section 6 concludes with some open questions.

2. Related results.

2.1. Previous work. The hypercube is known to contain, or nearly contain, many other natural structures as subgraphs. For example, the $n_1 \times n_2 \times \cdots \times n_k$ grid is a subgraph of the $\sum_{i=1}^{k} \lceil \log n_i \rceil$-dimensional hypercube.[1] Curiously, it is not a subgraph of any smaller hypercube. For example, a 3×5 grid is a subgraph of the 32-node hypercube, but it is not a subgraph of the 16-node hypercube. However, Chan [11] has recently shown that every $n_1 \times n_2$ grid can be embedded in a $\lceil \log n_1 n_2 \rceil$-dimensional hypercube with dilation no more than 2, and has further shown that the $n_1 \times \cdots \times n_k$ grid can be embedded one-to-one in the $\lceil \log n_1 n_2 \cdots n_k \rceil$-dimensional hypercube with dilation $O(k)$ [12].

The $(N-1)$-node complete binary tree is not a subgraph of the N-node hypercube,[2] although it can be embedded with dilation 2 since the N-node two-rooted complete binary tree (see Fig. 1) is a subgraph of the N-node hypercube [7], [18]. As a consequence, the $(N-1)$-node complete binary tree is a subgraph of the $2N$-node hypercube [30].

Even more complex and computationally powerful structures can be efficiently embedded within the hypercube. For example:

1. Leighton [22] showed that meshes of two-rooted trees are subgraphs of the hypercube; and Efe [13] showed that the mesh-of-trees is a subgraph of the hypercube;

[1] In this paper all logarithms are base 2.

[2] Both the tree and the hypercube are bipartite graphs. While the bipartite node sets for the hypercube are equal in size, they differ by a factor of 2 in the complete binary tree.

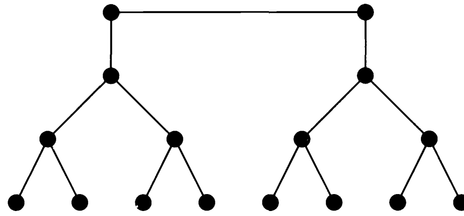


FIG. 1. *The two-rooted complete binary tree.*

2. Stout [28] showed that pyramid graphs can be embedded with dilation 2 and minimal expansion in the hypercube. Ho and Johnsson [19] improved this result to dilation 2 and congestion 2; and
3. Greenberg, Heath, and Rosenberg [17] showed that every FFT network is a subgraph of the smallest hypercube that can contain it. They also showed that the same result holds for butterfly networks of even order; butterflies of odd order can be embedded with dilation 2 in the smallest hypercube that can contain them.

In fact, the only bounded-degree graphs known not to have efficient embeddings in the hypercube are random graphs and expander-based graphs. In §4 we show that any constant-expansion, one-to-one embedding of such a graph into the hypercube must have dilation $\Omega(\log N)$, the worst possible (to within constant factors) since the N-node hypercube has diameter $\log N$.

We conjecture that the shuffle-exchange and deBruijn graphs are examples of bounded-degree graphs that cannot be embedded one-to-one into the hypercube with constant dilation and expansion. We know of no lower bounds for either case. We do not even know whether or not every bounded-degree planar graph can be embedded with constant dilation and expansion.

2.2. Extensions and subsequent work. This paper extends the results communicated earlier by the authors [5] when the node-congestion was not known to be $O(1)$. Our earlier communication [5] has since led to developments along a number of different directions, some of which we mention below.

Building on an earlier version [5], Monien and Sudborough [27] claim to have reduced the constants for dilation over our construction, but do not consider either node- or edge-congestion. For example, they claim that every N-node binary tree can be embedded with expansion 1 and dilation 5, or with dilation 3 and expansion $O(1)$. Whether or not any binary tree actually requires dilation 3 is open. Wagner [29] showed that every N-node binary tree is a subgraph of the hypercube with $O(N \log N)$ nodes. It is not known if every binary tree is a subgraph of an $O(N)$-node hypercube. Mayr [26] examined parallel algorithms which efficiently compute our embeddings.

The techniques in this paper were extended to embeddings within the butterfly, and related, networks. For example, the N-node complete binary tree can be embedded with constant expansion and dilation within the butterfly network [4] and, as a consequence of results in this paper, every N-node binary tree can be embedded one-to-one with constant expansion and $O(\log \log N)$ dilation and congestion within the butterfly network.

In [6] the authors applied the techniques used in this paper to construct a bounded-degree N-node graph that contains all N-node binary trees as spanning subgraphs.

This provided the first known example of a bounded-degree graph which is *universal* for all binary trees. Using an entirely different approach, Friedman and Pippenger [14] proved that every N-node bounded-degree tree is a (not necessarily spanning) subgraph of an $O(N)$-node expander graph.

Leighton and Malitz [23] constructed examples of N-node graphs with $O(N^{\alpha})$-separators for which every one-to-one constant-expansion embedding into the hypercube must have dilation at least $\Omega(\alpha \log N / - \log \alpha)$. Thus, for example, there exist graphs with $O(\sqrt{N})$-separators for which every constant-expansion one-to-one embedding must have dilation at least $\Omega(\log N)$. This implies that embedding strategies based on separators alone cannot be used to obtain constant-dilation embeddings for planar graphs even if such embeddings exist.

Bhatt and Cai [3] considered the problem of maintaining dynamically evolving trees on the hypercube. They showed, among other results, that a simple randomized embedding technique guarantees dilation $O(\log \log N)$ and, with high probability, $O(1)$ max-load to maintain an N-node dynamic tree on the N-node hypercube. Leighton, Newman, Ranade, and Schwabe [24] improved this result to dynamic embeddings with $O(1)$ dilation, node-congestion, and edge-congestion while maintaining load balance.

The results mentioned above are all concerned with embedding bounded-degree graphs within hypercubes. For such graphs, embeddings with constant dilation and congestion utilize only $O(N)$ out of the $\frac{1}{2}N \log N$ communication edges of the hypercube. Greenberg and Bhatt [16] and Aiello, Leighton, Maggs, and Newman [1] extend these embeddings to multiple-path embeddings in which each edge of the host graph is mapped to multiple, edge-disjoint, short paths in the hypercube.

Along a different direction, Koch et al. [21] extend the study of graph embeddings to examine work-preserving emulations among different interconnection networks.

3. Definitions. Before proceeding with our results, we need a few definitions. An *embedding* $\langle \phi, \rho \rangle$ of a graph $G = (V_G, E_G)$ into a graph $H = (V_H, E_H)$ is defined by a mapping ϕ from V_G to V_H, together with a mapping ρ that maps each edge $(u, v) \in E_G$ onto a path $\rho(\phi(u), \phi(v))$ in H that connects $\phi(u)$ and $\phi(v)$. The *load* on a node $v \in H$ is the number of nodes of G that are mapped onto v; the *max-load* of an embedding is the maximum load over all nodes of H. The *expansion* of an embedding is the ratio of the size of V_H to the size of V_G.

The *dilation* of the edge (u, v) under $\langle \phi, \rho \rangle$ equals the length of the path $\rho(\phi(u), \phi(v))$ in H; the *dilation* of an embedding $\langle \phi, \rho \rangle$ is the maximum dilation of an edge in G. The *congestion* of an edge e_H in H under $\langle \phi, \rho \rangle$ equals $|\{e \in E_G : \rho(e) \text{ contains } e_H\}|$; the *congestion* of an embedding $\langle \phi, \rho \rangle$ is the maximum congestion of any edge in H. The congestion of a node $v_H \in H$ under $\langle \phi, \rho \rangle$ equals $|\{e \in E_G : \rho(e) \text{ contains } v_H \}|$; the *node-congestion* of an embedding $\langle \phi, \rho \rangle$ is the maximum congestion of any node in H.

We are interested in efficient embeddings of one family $\mathcal{G}$ of graphs into another family $\mathcal{H}$ of graphs. The families $\mathcal{G}$ and $\mathcal{H}$ are parameterized by the number of nodes they contain. By an expansion $e(N)$, dilation $d(N)$ embedding of $\mathcal{G}$ into $\mathcal{H}$ we formally mean a set of embeddings where every N-node graph $G_N \in \mathcal{G}$ is embedded into $H_{Ne(N)} \in \mathcal{H}$ with dilation no greater than $d(N)$. Embeddings for which the expansion and dilation do not grow with N are of particular interest. In what follows, we always mean embeddings among families of graphs, although we do not always make that explicit.

4. A lower bound. Not all bounded-degree graphs can be embedded one-to-one within hypercubes with small dilation and expansion. In particular, in what follows we show that every constant-expansion, one-to-one embedding of expander graphs [25] within hypercubes must have dilation $\Omega(\log N)$, the worst possible. One property of the family of expander graphs is that the removal of any m nodes from an N-node expander graph, $m \leq N/2$, requires that at least αm edges be cut, where $\alpha > 0$ is a constant independent of N.

PROPOSITION 4.1. *Every constant-expansion, one-to-one embedding of the family of expander graphs into the family of hypercubes requires dilation* $\Omega(\log N)$.

Proof. In fact, we will prove that the average dilation grows as $\Omega(\log N)$. Consider an embedding of an N-node expander G in the $2^k N$-node hypercube, where k is a nonnegative constant. Partition the set of $k + \log N$ dimensions of the hypercube into $t = \lfloor (k + \log N)/(k+1) \rfloor$ subsets $S_i, 1 \leq i \leq t$, of $k+1$ dimensions each, and possibly one more set with fewer than $k + 1$ dimensions.

We first count the number of times edges of G traverse hypercube edges that lie in one of the dimensions of $S_i, 1 \leq i \leq t$. The removal of hypercube edges in dimensions within S_i splits the hypercube into 2^{k+1} blocks, each with $N/2$ nodes. This also splits the nodes of G into 2^{k+1} blocks, one of which contains at least $N/2^{k+1}$ nodes, and at most $N/2$ nodes. Since G is an expander, this means that at least $\alpha N/2^{k+1}$ edges of G ($\alpha > 0$ is some constant independent of N) each traverse at least one dimension of S_i. This is true for all i so the total number of dimension traversals is at least $t\alpha N/2^{k+1} = \Omega(N \log N)$. Because the number of dimension traversals is a lower bound on the sum of dilations of all edges, we have that the average dilation, and hence the dilation, is $\Omega(\log N)$. □

5. Embedding binary trees in thistle trees. To embed an arbitrary N-node binary tree T within the hypercube, we proceed in two stages. In the first stage, T is decomposed and efficiently embedded within the N-node *thistle tree.* The next section gives efficient embeddings of thistle trees within hypercubes; this second stage induces an efficient embedding of T in the hypercube.

Before defining thistle trees, however, we first present results on tree decomposition. These results are based on combinatorial techniques developed previously for VLSI layout [8], [9] and for constructing universal graphs for trees [6]. In particular, we will use a minor variant of the decomposition lemma from [6]. As mentioned in [8], the decomposition can be obtained in time polynomial in the size of T.

We begin with the notion of k-color bisectors. Suppose that every node of a graph G is colored with one of k colors. Further, let S be a set of nodes of G whose removal partitions the remaining nodes into two equal (to within one) subsets, both containing equal (again, to within one) numbers of nodes of each color, and such that there is no edge in G connecting a node from one subset to the other. Such a set S is called a k-color bisector of G.[3] Finally, we note that every N-node binary forest has a k-color bisector of size less than $k \log N$ [8], [9].

The following lemma is fundamental to our result. In what follows, the *depth* of a node in a tree is defined to be the distance from the root to that node; the root is at depth 0. In an N-node complete binary tree, a node at depth d is said to be at *level* $\log N - d$; leaves are at level 1.

[3] An immediate consequence of the definition is that a k-color bisector S for G can be extended into a k-color bisector $S' \supseteq S$ of any larger size, by removing nodes of the same color one by one, and alternating between the two separated subsets.

LEMMA 5.1. *Every N-node binary tree T can be mapped (many-one) to the level-$(\log N - 1)$ complete binary tree C so that* (a) *exactly* $6\log(N/2^t) + 18$ *nodes of T are mapped onto a node of C at depth $t < \log N - 7$, and at most* 60 *nodes of T are mapped to any node at depth $t = \log N - 7$, and no nodes of T are mapped at greater depth, and* (b) *any two nodes adjacent in T are mapped to nodes at most distance* 3 *apart in C. Furthermore, for every node of C, the numbers of nodes of T embedded within its two subtrees differ by at most* 1.

Remark. Lemma 5.1 is almost identical to Lemma 1 in [6]; the main difference being that the "exactly" in condition (a) above is replaced by "at most" in [6]. The proof remains almost identical, with the difference that whenever the proof in [6] uses bisectors of smaller size than stated in condition (a), we invoke the previous footnote to extend the bisector to the required size. By counting the number of nodes of T mapped at different depths, one can show that nodes of C at depths 0 through $\log N - 8$ are filled exactly as required, and at depth $(\log N - 7)$ each remaining subgraph has less than 60 nodes. The details are straightforward and are left to the reader.

For our purposes we will need to modify the above embedding slightly. Suppose that each node of C at level i (depth $\log N - 1 - i$) has maximum capacity i; i.e., at most i nodes of T can be placed at a level-i node of C. The number of nodes of T placed at nodes of depth $\log N - 8$ or less of C by Lemma 5.1 exceeds their capacity. In contrast, nodes at depth $\log N - 7$ and greater in C are assigned fewer nodes of T than their capacity allows. The following lemma states that we can perturb the mapping of T slightly so that capacity constraints are satisfied at every node of C, and without greatly increasing the distances between nodes adjacent in T.

LEMMA 5.2. *Every N-node binary tree T can be mapped (many-one) to the level-$(\log N - 1)$ complete binary tree C so that* (a) *at most i nodes of T are mapped onto a level-i node of C, and* (b) *any two nodes u and v that are adjacent in T are mapped to nodes U and V in C whose least common ancestor is at most distance* 8 *from each of U and V.*

Proof sketch. Given the embedding of Lemma 5.1, at each node of C make an ordered list of the nodes of T that are embedded there. Starting with the root, we "push" excess nodes of T down to lower levels as follows: when a node b is ready for "pushing," we fill b to capacity with the appropriate number of the "leftmost" vertices in the ordered list of nodes currently at b. The remaining nodes on the list are divided into two equal (to within 1) sublists and appended at the left end of the ordered lists for the children, b_0 and b_1, of b. The nodes b_0 and b_1 are now ready to be "pushed."

To establish that the above procedure maps every node of T within C (i.e., that no nodes of T are pushed out of C), we show inductively that the total number of nodes of T assigned to the subtree rooted at any node of C cannot exceed the total capacity of the subtree. Initially, this is true at the root. Suppose this is true at node b when it is ready to be pushed. We claim that after b has been pushed, the inductive hypothesis holds at each of the subtrees rooted at b_0 and b_1. Before b is pushed, the number of nodes of T mapped in the subtrees rooted at b_0 and b_1 are equal (to within one). Furthermore, equal (to within one) numbers of nodes are pushed onto b_0 and b_1 when b is pushed. The result is that the total numbers of nodes of T within the two subtrees remains equal (to within one) after b has been pushed. Therefore, if the capacity of either b_0 or b_1 is violated after b is pushed, it must be the case that the capacity of b was violated before b was pushed; this contradicts the inductive

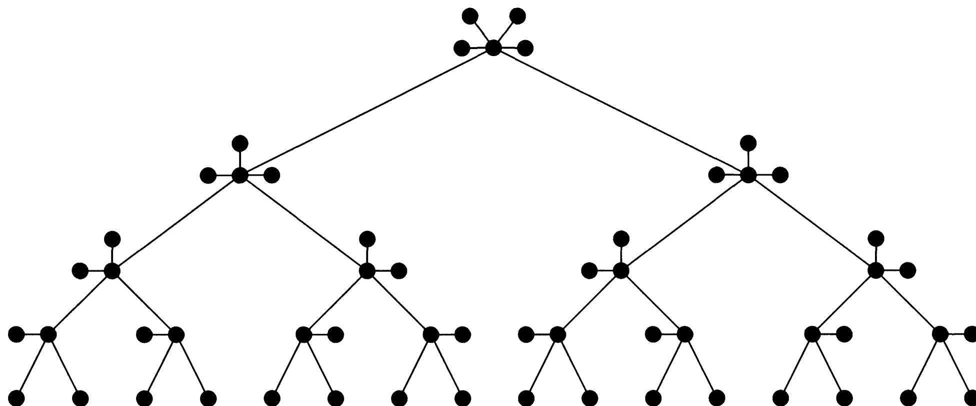


FIG. 2. *The thistle tree* T_5.

hypothesis.

For the second part of Lemma 5.2 we need to bound how far the push procedure can force a node of T to ripple down the tree. This is a simple calculation; the result is that the number of nodes assigned by Lemma 5.1 to depth at most ℓ in C is less than the total capacity of nodes with depth at most $\ell + 5$. This means that a node of T will be reassigned to a node of C with depth at most 5 greater than by Lemma 5.1. This suffices to guarantee part (b) of Lemma 5.2. □

5.1. Thistle trees. The decomposition of Lemma 5.2 motivates the definition of thistle trees. The thistle tree T_h is obtained by starting with a complete binary tree of $2^h - 1$ nodes and attaching to each level-i node, $1 \leq i \leq h$, $i - 1$ additional leaves called *thistles*. The thistle tree T_5 is shown in Fig. 2. The thistle tree T_k (of depth $k - 1$) has $2^{k+1} - k - 2$ nodes. For convenience we assume that our binary trees have size $N = 2^{k+1} - k - 2$. This assumption will be removed later.

DEFINITION. If u is a thistle adjacent to node w, then we call w the *central node* of u, and denote $X(u) = w$. If u is not a thistle node, then it is a central node and we define $X(u) = u$.

6. Embeddings in the hypercube.

THEOREM 6.1. *Every N-node binary tree can be embedded one-to-one in a hypercube with expansion* 1, *dilation* $O(1)$, *and congestion* $O(1)$.

Proof. The decomposition of Lemma 5.2 is invoked to embed an arbitrary N-node binary tree one-to-one within a thistle tree. Map the nodes of T that are embedded within the same internal node in Lemma 5.2 onto distinct thistles adjacent in the thistle tree, with one node of T per thistle. This gives an embedding of T in the thistle tree with expansion 1 and max-load 1.

It remains to embed the N-node thistle tree within the hypercube. The next section investigates embeddings of thistle trees within hypercubes. We then complete the proof of Theorem 6.1 by showing that in the induced embedding both node- and edge-congestion are $O(1)$.

6.1. The inorder embedding of complete binary trees. There is a very natural way of embedding complete binary trees within hypercubes. As may be seen in Fig. 3, an inorder labeling of the 15-node tree yields an expansion-1, dilation-2

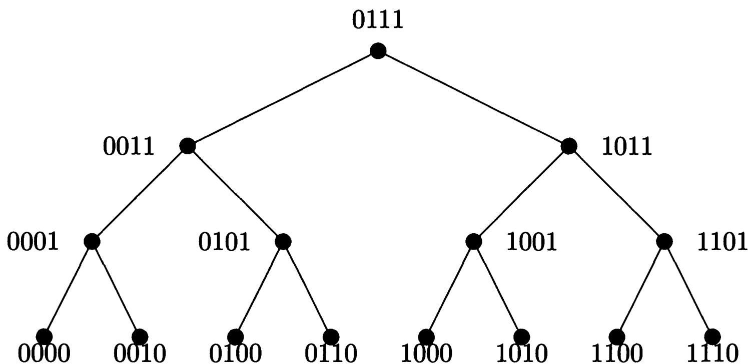


FIG. 3. *The inorder embedding of a complete binary tree.*

embedding. Each node v in the tree is labeled with the $\log N$-bit binary representation $b(v)$ of its inorder number; the hypercube address of v, $\phi(v)$, is defined to equal $b(v)$. This embedding scales to larger trees. In general, the ith leftmost node at level k ($k \geq 1, i \geq 0$) has inorder number $i2^k + 2^{k-1} - 1$; its left child (the $2i$th node at level $k-1$) has inorder number $i2^k + 2^{k-2} - 1$; and its right child has inorder number $i2^k + 2^{k-1} + 2^{k-2} - 1$. From this it follows that the inorder label of the left child of every node at level k differs from the inorder label of the node in bit position $k-2$ while the right child differs in bit positions $k-1$ and $k-2$. In other words, every left-edge (between a node and its left child) has dilation 1, and every right-edge has dilation 2.

We associate with each node u in the complete binary tree a path, $\tau(u)$, the *trace* of u, which starts at the left child of u and follows the rightmost path down the tree to a leaf. It is easily seen that every node w of the tree (except along the rightmost path from the root) lies in the trace of exactly one other node u. If a node w lies in $\tau(u)$, then we call u the *source* of node w.

The inorder numbering of a complete binary tree C of $2^n - 1$ nodes also has the following useful properties which can be verified in a straightforward manner.

1. The descendants of an internal node u that lie ℓ levels below u occupy an ℓ-dimensional subcube. The descendants that lie no more than ℓ levels below u reside in an $(\ell + 1)$-dimensional subcube.
2. For every i, each node u at level i is adjacent in the hypercube to source(u) along an edge in dimension i. Therefore, the nodes in $\tau(u)$ are adjacent to u along dimensions $i-1, i-2, \cdots, 1$.
3. If S is the set of descendants of u that lie at most distance m away from u, and if u is at level i of C, then the set of nodes at level j ($j < i$) which are in the trace of nodes in S lie within an $(m+1)$-dimensional subcube. As j varies, these subcubes are disjoint but are defined by the same set of $m+1$ dimensions for all j.

6.2. Embedding the thistle tree. We embed a height-h thistle tree into a height-h complete binary tree as follows: embed the central node of each thistle onto its counterpart in the complete binary tree, and embed the $i-1$ thistles connected to a central node u at height i one-to-one onto the $i-1$ nodes in the trace $T(u)$. The properties of the inorder embedding mentioned in the previous section induce an

embedding of the height-h thistle tree into a 2^{h+1}-node hypercube with dilation 2, max-load 2, and expansion $\frac{1}{2}$. Of the two thistle-tree nodes mapped to one hypercube node, one is a central node and the other a thistle. We obtain a one-to-one embedding by first constructing a 2^{h+2}-node hypercube by taking two cubes of half the size. The entire thistle tree lies in one of the half-size cubes. We project each thistle node over to the corresponding empty hypercube node across the matching that connects the two half-size cubes to obtain an embedding with dilation 2, expansion 1, and max-load 1.

For convenience, we fix the following notation. Let V_T, V_{TT}, V_{CBT}, and V_H be, respectively, the node sets of the binary tree, thistle tree, complete binary tree, and the hypercube, so that $|V_T| = |V_{TT}| \leq |V_H| = N$, and $|V_{CBT}| = (N-1)/2$. The maps between these node sets are named as follows:

$$\begin{array}{llll} \alpha & : & V_T & \mapsto & V_{TT}, \\ \beta & : & V_{TT} & \mapsto & V_{CBT} \text{ (not one-to-one)}, \\ \gamma & : & V_{TT} & \mapsto & V_H, \\ \phi & : & V_T & \mapsto & V_H \ (\phi = \gamma \circ \alpha). \end{array}$$

We now proceed with the proof of Theorem 6.1. In the embedding α, nodes u and v adjacent in T are mapped to nodes $\alpha(u)$ and $\alpha(v)$ whose central nodes are $X(\alpha(u))$ and $X(\alpha(v))$. The images $\gamma(X(\alpha(u)))$ and $\gamma(X(\alpha(v)))$ lie within a nine-dimensional subcube. How far apart can the images $\phi(u)$ and $\phi(v)$ be? In the worst case, $\phi(u) = \gamma(\alpha(u))$ and $\phi(v) = \gamma(\alpha(v))$ can be distance 1 away from $\gamma(X(\alpha(u)))$ and $\gamma(X(\alpha(v)))$, respectively, so the distance between $\phi(u)$ and $\phi(v)$ can be no greater than 11.

We have to find an assignment of paths within the hypercube to tree edges such that both node- and edge-congestion in the hypercube are $O(1)$. In the general case, suppose that we have to route tree edge (u, v) between hypercube nodes $\phi(u)$ and $\phi(v)$.

Let $U = \gamma(X(\alpha(u)))$ and $V = \gamma(X(\alpha(v)))$ denote the images of the central nodes of u and v. Let D_{UV} be the set of hypercube dimensions in which U and V differ. Further, suppose that $\beta(\alpha(u))$ and $\beta(\alpha(v))$ are at levels ℓ_u and ℓ_v of the complete binary tree, respectively, so that $\phi(u)$ and U differ in dimension ℓ_u, and $\phi(v)$ and V differ in dimension ℓ_v. Assume that $\ell_u < \ell_v$ (the case when they are equal is covered as a simpler subcase).[4]

The naive way (Fig. 4) to route edge (u, v) is to follow dimension ℓ_u from $\phi(u)$ to U, follow images of thistle-tree edges (within a nine-dimensional subcube) to V, and finally follow dimension ℓ_v to reach $\phi(v)$. The problem with this scheme is that the congestion along images of thistle-tree edges can be as large as $\Omega(\log N)$.

We can make both node- and edge-congestion $O(1)$ by traversing the dimensions in a different order, in three stages. As indicated in Fig. 5, in Stage 1 we follow a path which traverses dimensions in D_{UV} (in any order within the corresponding nine-dimensional subcube) from $\phi(u)$ to the node U_1. Observe that $\beta(\gamma^{-1}(\phi(u)))$ and $\beta(\gamma^{-1}(U_1))$ lie at the same level in the complete binary tree. In Stage 2 we follow dimension ℓ_v from U_1 to U_2, and in Stage 3 we follow dimension ℓ_u from U_2 to reach $\phi(v)$. Because $\ell_u < \ell_v$, it follows that $\beta(\gamma^{-1}(U_2))$ in the complete binary tree lies in the trace of $\beta(\alpha(v))$, and also that $\beta(\gamma^{-1}(U_2)), \beta(\gamma^{-1}(U_1))$, and $\beta(\alpha(u))$ all lie at level ℓ_u of the complete binary tree.

[4] The general case includes degenerate cases such as, for example, when $\alpha(u)$ or $\alpha(v)$ is a central node. We do not explicitly mention these cases.

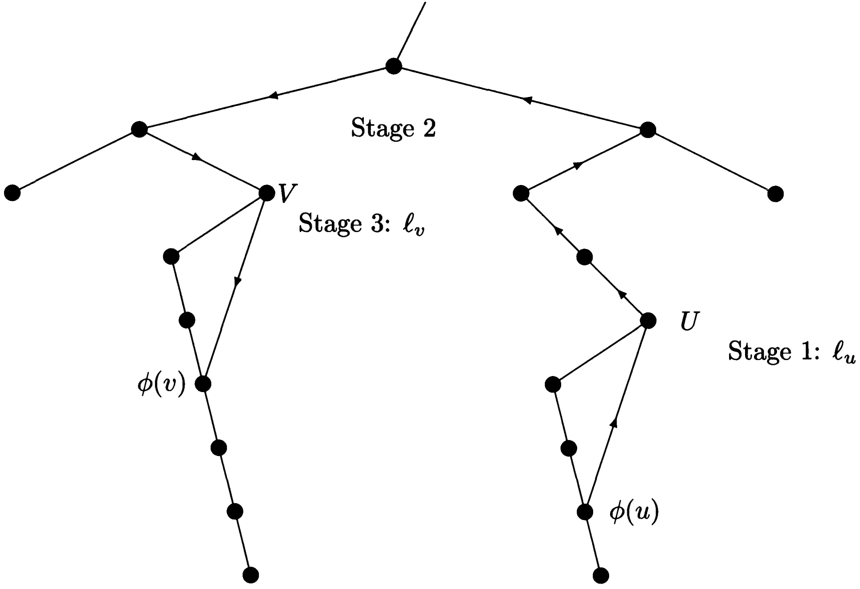


FIG. 4. *The naive scheme for routing paths.*

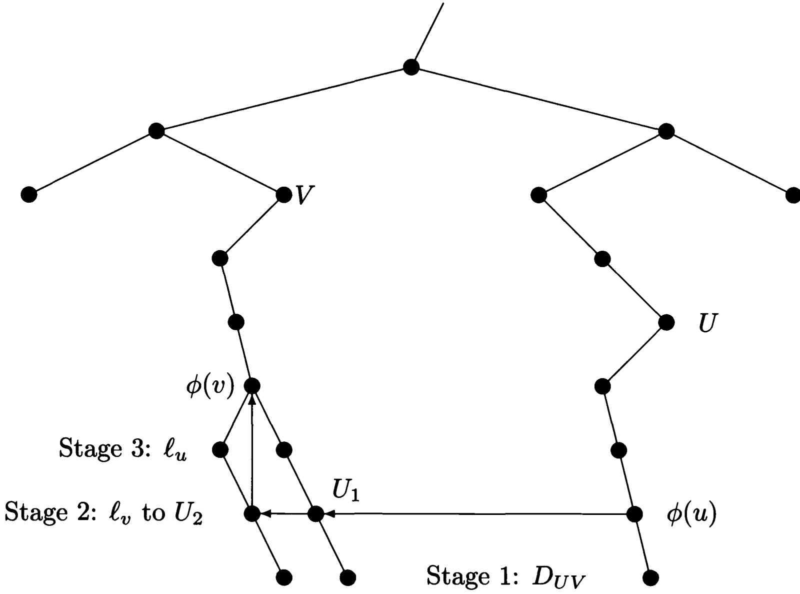


FIG. 5. *The modified scheme.*

With this modified routing, we claim that both node- and edge-congestion are $O(1)$. Consider the set of routes in Stage 1. By property 3 of inorder embeddings, the path from $\phi(u)$ to U_1 lies within a nine-dimensional subcube; the sources of their pre-images in the complete binary tree can be no further than eight levels away from their lowest common ancestor. Now u can be adjacent in T to another node w, in which case the route from $\phi(u)$ to W_1 in Stage 1 (define W, W_1, W_2 accordingly as for U, U_1, U_2) will lie within a nine-dimensional subcube defined by a set D_{UW}. The two sets D_{UV} and D_{UW} can be different so that the two routes starting at $\phi(u)$ can

lie in different nine-dimensional subcubes. However, every route starting at $\phi(u)$ is restricted to lie in one of only nine possible nine-dimensional subcubes because there are only eight choices for the lowest common ancestor, and each choice fixes a distinct nine-dimensional subcube. Because each such subcube has $O(1)$ nodes, and each node in a binary tree is the origin of up to three routes, both node- and edge-congestion in Stage 1 are $O(1)$.

Next consider Stage 2, in which each route is a single step. Because the node-congestion in Stage 1 is $O(1)$, we are guaranteed that each node in Stage 2 is the origin of only $O(1)$ messages. This suffices to guarantee that the edge-congestion in Stage 2 is $O(1)$. Observe that in Stage 3 all edges leaving nodes whose pre-images in the complete binary tree are at level ℓ_u are routed along dimension ℓ_u; therefore, at the end of Stage 2, routes terminating at U_2 correspond to edges incident to tree nodes which are mapped either onto U_2 or onto $\phi(v)$. Since each tree node has bounded degree, the node-congestion in Stage 2 is $O(1)$. In Stage 3 the node- and edge-congestion are $O(1)$ because of bounded-node degrees. Overall, therefore, both node- and edge-congestion are $O(1)$.

Finally, when the size N of the tree lies between $2^{k+1} - k - 2$ and 2^{k+1}, we first remove $N - 2^{k+1} + k + 2$ nodes, and embed the subtree of $2^{k+1} - k - 2$ nodes. Next, we use the fact [15] that within an m-dimensional hypercube m node-disjoint paths can be found to connect any m source nodes to any m sinks. By creating a sink which is vacant and sources wherever one of the removed nodes must be embedded, $k + 1$ additional nodes can be embedded by percolating nodes along the flow paths. This increases dilation, node-, and edge-congestion by $O(1)$. The last node can be inserted similarly, and the overall dilation, node-, and edge-congestion remain $O(1)$. □

7. Extensions and conclusions. This paper gives simulations of tree structures in the hypercube. The decomposition lemma (Lemma 5.2) for binary trees also provides optimal embeddings of binary trees within other networks. For example, we can show that every N-node binary tree can be embedded within an N-node complete binary tree with expansion 1 and dilation $O(\log\log N)$. By embedding a complete binary tree within the shuffle-exchange graph with expansion 1 and dilation 2, we obtain $O(\log\log N)$ dilation for arbitrary trees embedded within the shuffle-exchange graph. We have not yet determined whether or not these bounds are optimal to within constant factors.

All of our results on embeddings within the hypercube extend to arbitrary graphs of bounded degree with $O(1)$ separators. While our simulations are optimal to within constant factors, there is much room for reducing the overhead in expansion and dilation further. It would be satisfying to discover simpler ways to embed binary trees in the hypercube. For example, we do not know of any specific binary tree that cannot be embedded in the hypercube either with expansion 1 and dilation 2 or with expansion 2 and dilation 1.

Acknowledgments. Thanks to David Greenberg for helpful discussions and for his careful reading of the manuscript. Thanks also to Lennart Johnsson for early suggestions.

REFERENCES

[1] W. Aiello, F. Leighton, B. Maggs, and M. Newman, *Fast algorithms for bit-serial routing on a hypercube*, in Proc. Second Annual ACM Symposium on Parallel Algorithms and Architectures, 1990, pp. 55–62.

[2] F. Berman and L. Snyder, *On mapping parallel algorithms into parallel architectures*, J. Parallel Distrib. Computing, 4 (1987), pp. 439–458.

[3] S. Bhatt and J. Cai, *Take a walk, grow a tree*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, 1988, pp. 469–478.

[4] S. Bhatt, F. Chung, J. Hong, F. Leighton, B. Obrenic, A. Rosenberg, and E. Schwabe, *Optimal emulations by butterfly-like networks*, J. Assoc. Comput. Mach., to appear. (A preliminary version, *Optimal simulations by butterfly networks*, appears in Proc. 20th Annual ACM Symposium on Theory of Computing, 1988, pp. 192–204.)

[5] S. Bhatt, F. Chung, F. Leighton, and A. Rosenberg, *Optimal simulations of tree machines*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 274–282.

[6] ———, *Universal graphs for bounded-degree trees and planar graphs*, SIAM J. Discrete Math., 2 (1989), pp. 145–155.

[7] S. Bhatt and I. Ipsen, *How to embed trees in hypercubes*, Tech. Report RR-443, Department of Computer Science, Yale University, New Haven, CT, 1985.

[8] S. Bhatt and F. Leighton, *A framework for solving* VLSI *graph layout problems*, J. Comput. System Sci., 28 (1984), pp. 300–343.

[9] S. Bhatt and C. Leiserson, *How to assemble tree machines*, in Advances in Computing Research 2, F. Preparata, ed., JAI Press, Greenwich, CT, 1984, pp. 95–114.

[10] S. Bokhari, *On the mapping problem*, IEEE Trans. Comput., C-30 (1981), pp. 207–214.

[11] M. Chan, *Dilation-2 embeddings of grids into hypercubes*, in Proc. IEEE Internat. Conference on Parallel Processing, 1988, pp. 295–298.

[12] ———, *Embedding of d-dimensional grids in optimal hypercubes*, in Proc. First ACM Symposium on Parallel Algorithms and Architectures, 1989, pp. 52–57.

[13] K. Efe, Personal communication, 1990.

[14] J. Friedman and N. Pippenger, *Expanding graphs contain all small trees*, Combinatorica, 7 (1987), pp. 71–76.

[15] D. Greenberg, *Minimum expansion embeddings of meshes in hypercubes*, Tech. Report RR-535, Department of Computer Science, Yale University, New Haven, CT, 1987.

[16] D. Greenberg and S. Bhatt, *Routing multiple paths in hypercubes*, in Proc. Second ACM Symposium on Parallel Algorithms and Architectures, 1990, pp. 45–54.

[17] D. Greenberg, L. Heath, and A. Rosenberg, *Optimal embeddings of butterfly-like graphs in the hypercube*, Math. Systems Theory, 23 (1990), pp. 61–77.

[18] I. Havel and P. Liebl, *Embedding the polytomic tree into the n-cube*, Časopis Pěst. Mat., 98 (1973), pp. 307–314.

[19] C. Ho and S. Johnsson, *Embedding hyper-pyramids into hypercubes*, Tech. Report RR-667, Department of Computer Science, Yale University, New Haven, CT, 1988.

[20] S. Johnsson, *Communication efficient basic linear algebra computations on hypercube architectures*, J. Parallel Distrib. Comput., 4 (1987), pp. 133–172.

[21] R. Koch, F. Leighton, B. Maggs, S. Rao, and A. Rosenberg, *Work-preserving emulations of fixed-connection networks*, in Proc. 21st Annual ACM Symposium on Theory of Computing, 1989, pp. 227–240.

[22] F. Leighton, *Lecture notes on theory of parallel and distributed computation*, Tech. Report LCS/RSS/1, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[23] F. Leighton and S. Malitz, *Separators and hypercube embeddings*, unpublished, 1990.

[24] F. Leighton, M. Newman, A. Ranade, and E. Schwabe, *Dynamic tree embeddings in hypercubes and butterflies*, in Proc. First ACM Symposium on Parallel Algorithms and Architectures, 1989, pp. 224–234.

[25] A. Lubotzsky, R. Philips, and P. Sarnak, *Ramanujan conjecture and explicit constructions of expanders*, in Proc. 18th Annual ACM Symposium on Theory of Computing, 1986, pp. 240–246.

[26] E. Mayr, Personal communication, 1987.

[27] B. Monien and I. Sudborough, *Simulating binary trees on hypercubes*, in VLSI Algorithms and Architectures, Lectures Note in Computer Science 319, Springer-Verlag, Berlin, New York, 1988, pp. 170–180.

[28] Q. Stout, *Hypercubes and pyramids*, in Pyramidal Systems for Computer Vision, V. Cantoni and S. Levialdi, eds., Springer-Verlag, Berlin, New York, 1986.

[29] A. Wagner, *Embedding trees in the hypercube*, Tech. Report DCS/204-87, University of Toronto, 1987.

[30] A. Y. Wu, *Embedding of tree networks into hypercubes*, J. Parallel and Distrib. Comput., 2 (1985), pp. 238–249.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 163–180, February 1992
©1992 Society for Industrial and Applied Mathematics
013

ON THE EXACT LOCATION OF STEINER POINTS IN GENERAL DIMENSION*

TIMOTHY LAW SNYDER†

Abstract. The Steiner Problem is to form a minimum-length tree that contains a given set of points, where augmentation of the point set with additional (Steiner) points is permitted. The Rectilinear Steiner Problem is one in which edge weights are determined by the L_1, or Manhattan, distance between points in $\mathbb{R}^d$.

Let S be a set of n points $\mathbb{R}^d$, where $d \geq 2$. By generalizing a planar theorem of Hanan (*SIAM J. Appl. Math.*, 14 (1966), pp. 255–265.) to all dimensions, a set of $O(n^d)$ points that is guaranteed to include all the Steiner points of a minimum rectilinear Steiner tree of S is constructed, complementing Hanan's result in $d = 2$.

The theorem here and its proof illuminate new combinatorial and geometrical facts about rectilinear Steiner trees; one immediate benefit is algorithms that are asymptotically faster than all known algorithms for the problem in dimensions three and greater. The theorem also yields a polynomial algorithm in all dimensions for a version of the problem known as the Rectilinear *k-Steiner* Problem.

Key words. rectilinear Steiner trees, L1 Metric, Hanan's Theorem

AMS(MOS) subject classifications. 51M05, 90B99, 68R10, 05C05

1. Introduction and main results. The purpose of this paper is to construct, given a point set S in any dimension d, a relatively small set of points that is guaranteed to contain all the Steiner points of a minimum rectilinear Steiner tree of S. This construction immediately yields faster algorithms for the Rectilinear Steiner Problem in general dimension.

Let S be a finite set of points (or vertices) in $\mathbb{R}^d$. A *Steiner tree* of S is a tree whose vertex set spans S. This means that a Steiner tree can contain vertices that do not belong to the original point set S; these vertices are called *Steiner points.* If we let $l(e) > 0$ be the weight (length) of an edge e connecting two distinct points of $\mathbb{R}^d$, then we can define a *minimum Steiner tree of* S as a tree $T^\star$ satisfying

$$\sum_{e \in T^\star} l(e) = \min_T \left\{ \sum_{e \in T} l(e) : T \text{ is a Steiner tree of } S \right\}. \tag{1.1}$$

The *Steiner Problem*, named after an eighteenth century mathematician and significantly popularized in this century by Courant and Robbins (1941), is: given a point set $S \subset \mathbb{R}^d$ and a method for assigning edge weights to the complete graph on S, find a minimum Steiner tree of S.

Edge weights in the Steiner Problem are often determined using a distance metric. If e joins the points $x = (x_1, x_2, \ldots, x_d)$ and $y = (y_1, y_2, \ldots, y_d) \in \mathbb{R}^d$, then the *rectilinear distance* $r(e)$ is defined by $r(e) = |x_1 - y_1| + |x_2 - y_2| + \cdots + |x_d - y_d|$. Geometrically interpreted, the edge e having rectilinear distance $r(e)$ consists of contiguous line segments that parallel the d coordinate axes.

Since the given point set in the Steiner Problem is a subset of $\mathbb{R}^d$, it is not obvious a priori that the problem can be solved in finite time, for the Steiner points

*Received by the editors September 17, 1990; accepted for publication (in revised form) February 11, 1991.

†Department of Computer Science, Georgetown University, Washington, DC 20057.

can occupy an infinite number of places in any region of $\mathbb{R}^d$. Unlike the Minimum Spanning Tree Problem, which forms a minimum-length tree on a given set of points *without* the ability to augment the point set with additional points, the Rectilinear Steiner Tree Problem is *NP*-complete for a discretized version of the L_1 metric (see Garey, Graham, and Johnson (1976) and Garey and Johnson (1977)).

By results contained in Melzak (1961) and the comprehensive paper of Gilbert and Pollak (1968), however, algorithms for the Euclidean Steiner Problem exist in all dimensions. Unfortunately, these methods are not metric-independent, hence they do not apply to the Rectilinear Steiner Problem (Hwang (1976)).

The only known algorithm for the rectilinear problem in general dimension is due to Sankoff and Rousseau (1975), who showed that, given a set of points and a topology of a Steiner tree spanning those points, where the topology includes some Steiner points whose locations are unknown, the location of the Steiner points can be found in polynomial time using dynamic programming. To find a minimum rectilinear Steiner tree (MRST) using this algorithm, one must enumerate all possible tree topologies for all possible numbers of Steiner points; this is costly. The algorithms that result from the theorem in this paper are asymptotically faster than the algorithm of Sankoff and Rousseau (1975), but the Sankoff–Rousseau algorithm is valid for any metric.

An algorithm for the Rectilinear Steiner Problem in dimension two was first established by Hanan (1966) using the following theorem.

HANAN'S THEOREM. *Let* $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\} \subset \mathbb{R}^2$. *Then, there exists a minimum rectilinear Steiner tree* T *of* S *such that, if* Q *is a Steiner point of* T, *then* $Q = (x_i, y_j)$, *for some* $1 \le i \le n$ *and* $1 \le j \le n$.

Hanan's Theorem tells us that if we construct a grid by drawing lines parallel to the x and y axes through each of the points of S, then there exists an MRST of S, each of whose Steiner points belongs to the set of intersection points of the lines that form the grid. Hanan's Theorem is used to yield a finite (though enumerative) solution to the problem in $d = 2$, for it makes finite the number of places at which a Steiner point can occur.

By generalizing Hanan's Theorem to all dimensions $d \ge 2$, given a set of points $S \subset \mathbb{R}^d$, we can construct a point set that is polynomial in size and is guaranteed to contain all the Steiner points of an MRST of S. One immediate application of the theorem is a faster algorithm for the Rectilinear Steiner Problem in all dimensions. This is done in §2, where we use our generalization of Hanan's Theorem to form a graph that is amenable to the graph-based algorithm of Dreyfus and Wagner (1972). We then use the theorem to form a simple algorithm for the *k-Steiner Problem*, which restricts the number of Steiner points to be k or less. The algorithm solves the rectilinear k-Steiner Problem in polynomial time for $k = O(1)$. This result complements the algorithm of Georgakopoulos and Papadimitriou (1987) for the Euclidean 1-Steiner Problem, and it is the first polynomial algorithm for the rectilinear problem in general dimension.

The generalization of Hanan's Theorem is nontrivial. An exhaustive analysis like Hanan's is impossible in higher and general dimension, where the consideration of all local topologies of a Steiner point becomes unwieldy, and where the degree of a Steiner point can be as large as $2d$, as opposed to the degrees 3 and 4 guaranteed in $d = 2$. Hence, to generalize Hanan's Theorem to higher dimensions requires a new understanding of Steiner trees. The proof is made possible only by some new combinatorial objects, namely, the wire codes, canonical forms for local topologies, and trombone wires introduced in the first half of this paper.

Because of its applications to VLSI and general circuit design, rectilinear distance

is often the metric of interest in the Steiner Problem, since a minimum rectilinear Steiner tree in dimension two represents the minimal cost of connection of electronic components when wiring is constrained to travel in strictly vertical and horizontal directions. For this reason, edges in an MRST are traditionally referred to as *wires*; we adopt this convention here.

In higher dimensions, the Steiner problem has applications in communication networks and three-dimensional circuit technologies. There is also some intriguing biological literature using Steiner trees in higher dimensions to analyze neural optimization in the brain (Cherniak (1989)) and to suggest historical evolutionary patterns for problems in phylogeny (see, e.g., Foulds, Hendy, and Penny (1979); Foulds and Graham (1982); Foulds (1984); and Gusfield (1990)).

Furthermore, the length of the minimum Steiner tree in all dimensions is linked to the length of other d-dimensional combinatorial objects, including the optimal traveling salesman tour; the minimum enclosing rectangle of the point set S for $d = 2$ (Hanan (1966) and Chung and Hwang (1979)); and the minimum spanning tree (Hwang (1976) and Bern (1988) for the rectilinear case and Chung and Graham (1981) (and others) for the Euclidean case).

This paper is organized as follows. Section 2 formally states our Generalized Hanan Theorem and our algorithms. The theorem, which is necessary to establish the algorithms' correctness, is proved in §§3–8.

Some of the objects and analyses used in the proof of the theorem are of independent interest. In §3, we begin to examine the local topology of a Steiner point in d dimensions by developing the notion of *wire codes*. Section 4 proves the existence of an MRST whose wire codes satisfy certain properties by assuming a particular canonical form; these properties are then used in §5 to further elucidate the local topology of a Steiner point. Section 6 introduces *trombone wires* and *trombone trees*, along with some of their properties, and §§7 and 8 conclude the proof by sliding the trombone wires into place such that our theorem is satisfied. Section 9 concludes with some remarks and open problems.

2. A Generalized Hanan Theorem and its algorithmic implications. In this section, we present our Generalized Hanan Theorem and the algorithms that immediately follow from it. In general dimension $d \geq 2$, we will find that given any point set $S \subset \mathbb{R}^d$, there exists an MRST of S, each of whose Steiner points lies at an intersection point of a grid formed by hyperplanes normal to the coordinate axes. We call this set of hyperplanes the *Hanan Grid* of S.

To formally define the Hanan Grid of S, let $x_1, x_2, \ldots,$ and x_d be the coordinate axes (or dimensions) of $\mathbb{R}^d$, and consider the point $P = (p_1, p_2, \ldots, p_d) \in S$. There exist d distinct hyperplanes containing P that are normal to the coordinate axes. For $1 \leq i \leq d$, we let $N(P, i)$ be the hyperplane passing through P that is normal to the x_i-axis. In other words,

$$N(P, i) = \{ (x_1, x_2, \ldots, x_d) \mid x_i = p_i \}, \quad \text{where } 1 \leq i \leq d. \tag{2.1}$$

We now define the Hanan Grid of S as

$$H(S) = \bigcup_{\substack{P \in S \\ 1 \leq i \leq d}} N(P, i), \tag{2.2}$$

i.e., the Hanan Grid $H(S)$ is formed by passing d hyperplanes normal to the coordinate axes through each member of S. The grid $H(S)$ consists of $d\,|S|$ (not necessarily distinct) hyperplanes.

For each subset of d mutually orthogonal hyperplanes of $H(S)$, i.e., for each subset of $H(S)$ of the form $\{N(P_1,1), N(P_2,2), \ldots, N(P_d,d)\}$, where, for $1 \leq i \leq d$, the points $P_i \in S$ are not necessarily distinct, there is a point at which the d hyperplanes intersect. We call the set of these intersection points $I_{H(S)}$. We note that $S \subset I_{H(S)}$ and $|S| \leq |I_{H(S)}| \leq |S|^d$.

We can now state our generalized theorem.

GENERALIZED HANAN THEOREM. *For any finite set of points $S \subset \mathbb{R}^d$, there exists a minimum rectilinear Steiner tree T of S such that, if Q is a Steiner point of T, then*

$$Q \in I_{H(S)}. \tag{2.3}$$

This generalization of Hanan's Theorem tells us that, given $S \subset \mathbb{R}^d$, there exists an MRST of S, all of whose Steiner points are intersection points of the Hanan Grid $H(S)$.

The Generalized Hanan Theorem allows us to use an algorithm of Dreyfus and Wagner (1972) to obtain a faster algorithm for the Rectilinear Steiner Problem in all dimensions. For purposes of comparison, we first note that, given n points, the algorithm of Sankoff and Rousseau (1975) must enumerate for each $0 \leq k \leq n-2$ all possible topologies of a tree on $n+k$ vertices. Since there exist exactly $(n+k)^{n+k-2}$ topologies of a tree of $n+k$ vertices (due originally to Cayley; see, e.g., Roberts (1984, p. 130), the running time of Sankoff and Rousseau's algorithm is

$$\begin{aligned} T_{SR}(n+k)\sum_{k=0}^{n-2}(n+k)^{n+k-2} &> (2n-2)^{2n-4}\,T_{SR}(2n-2) \\ &= O(4^{n-2}(n-1)^{2n-4}n^{c_1}), \end{aligned} \tag{2.4}$$

where c_1 is constant. Here, $T_{SR}(n+k)$ is the (polynomially bounded) time the Sankoff–Rousseau procedure takes to find the location of Steiner points given the topology of a tree on $n+k$ points. (The locations of the n given points are known.)

Dreyfus and Wagner's algorithm is designed for the Steiner Problem on graphs. Given a weighted graph G with t distinguished vertices called *terminals* and m vertices total, the Dreyfus–Wagner algorithm finds a minimum-weight subgraph (tree) of G that spans the set of terminals in $O(m3^t + m^2 2^t + m^3)$ time.

Given a point set $S \subset \mathbb{R}^d$, we can use the Generalized Hanan Theorem to form a graph problem amenable to the Dreyfus–Wagner algorithm that is equivalent to the problem of finding an MRST of S. Specifically, we let the intersection points of the Hanan Grid $H(S)$ be the vertices of the graph, with the vertices of S comprising the terminals to be spanned.

The edges of the graph are the line segments of each subset of $d-1$ mutually orthogonal intersecting hyperplanes of $H(S)$. Let $|S| = n$. Since $|I_{H(S)}| \leq n^d$, Dreyfus and Wagner's algorithm applied to the Hanan Grid runs in $O(n^d 3^n)$ time, since d is fixed. This algorithm is the fastest known algorithm for the Rectilinear Steiner Problem in general dimension.

As a second algorithmic application of the Generalized Hanan Theorem, we consider the *Rectilinear k-Steiner Problem*, which is to find an MRST T of a given point set under the restriction that T contains at most k Steiner points. The Euclidean version of the 1-Steiner problem in $d = 2$ was solved in quadratic time by Georgakopoulos and Papadimitriou (1987). The Generalized Hanan Theorem can be used to form a

simple "minimum spanning tree–based" algorithm that solves the k-Steiner Problem in polynomial time if $k = O(1)$. This algorithm enumerates all $\binom{n^d}{k}$ subsets $K \subset I_{H(S)}$ for each $0 \leq k \leq n-2$ and computes for each K the minimum spanning tree T_K of $K \cup S$. Over all trees T_K, a tree of minimum weight is an MRST of S. This algorithm has a running time of

$$T_{mst}(n+k) \sum_{k=0}^{n-2} \binom{n^d}{k},$$

where $T_{mst}(n+k)$ is the (polynomial) time necessary to compute a minimum spanning tree of $n+k$ points. From the bound

$$\binom{n^d}{k} < n^{dk}/k!,$$

it is easily seen that the algorithm solves the rectilinear k-Steiner Problem in polynomial time for any constant k, and does so even if we only bound k as $k = O(1)$. We remark that this algorithm is easily implemented once one has the minimum spanning tree code on hand.

The limitations of these algorithms and further implications of the theorem are discussed in §9.

To prove the Generalized Hanan Theorem and establish the correctness of our algorithms, we will analyze the local topology of a Steiner point, demonstrating that for every Steiner point Q of a certain MRST, we can find in each hyperplane $N(Q, i)$, where $1 \leq i \leq d$, a member of the original point set S. To achieve that end, we first must characterize the local topology of a Steiner point in general dimension; this is the topic of the next three sections.

3. Wire codes. The Generalized Hanan Theorem in §2 tells us that we can find all the Steiner points of a certain MRST at the intersection points of the Hanan Grid. To prove the theorem, we must first attempt to characterize the local topology of a Steiner point. We begin by generalizing some elementary observations of Hanan (1966), then defining wire codes, which will assist in describing the local topology and eventually proving our theorem.

Let $d(P)$ be the degree of the vertex P in an MRST. We have the following lemma, which contains generalizations of some elementary facts.

LEMMA 3.1. *If T is an* MRST *of $S \subset \mathbb{R}^d$ and $\overline{Q}$ is the set of Steiner points of T, then*

$$\begin{aligned} &\text{(i) } \textit{for all } Q \in \overline{Q},\ 3 \leq d(Q) \leq 2d; \\ &\text{(ii) } \textit{for all } P \in S,\ 1 \leq d(P) \leq 2d; \quad \textit{and} \\ &\text{(iii) } 0 \leq |\overline{Q}| \leq |S| - 2. \end{aligned} \tag{3.1}$$

Observations (3.1(i)) and (3.1(ii)) are obvious. For a proof of (3.1(iii)), one can consult Hanan (1966) or Gilbert and Pollak (1968) for dimension-two proofs; the proof in d dimensions follows. □

To examine the local topology of a Steiner point, we develop the notion of wire codes. Let w be a rectilinear wire incident with a Steiner point Q, and let $s(w)$ be the number of segments of w. The *wire code* of w, $c(w)$, is a sequence of $s(w)$ *symbol pairs* $\epsilon_j i_j$, where $1 \leq j \leq s(w)$. Formally,

$$c(w) = (\epsilon_1 i_1, \epsilon_2 i_2, \ldots, \epsilon_{s(w)} i_{s(w)}). \tag{3.2}$$

Each symbol pair of $c(w)$ represents a segment of the wire w in the order of its appearance, if we begin at Q and traverse w until we reach another vertex of the Steiner tree. The parameter ϵ_j represents the direction of travel and i_j represents the axis that the segment parallels; we use $\epsilon_j =$ " + " or $\epsilon_j =$ " − " and $i_j \in \{1, 2, \ldots, d\}$ as the allowable symbols.

We can completely specify w if we have information on the lengths of its segments. These are represented by the *distance code*

$$\rho(w) = (\rho_1, \rho_2, \ldots, \rho_{s(w)}), \tag{3.3}$$

where, for $1 \leq j \leq s(w)$, $\rho_j \in \mathbb{R}$ is the Euclidean length of the jth segment of w if we traverse w, beginning at Q.

Wire codes and distance codes are simply devices that chronicle some notes that might be taken by an electron traveling along a wire from a Steiner point Q. For each segment traversed, the electron notes its direction of travel by a symbol pair in $c(w)$, and it notes the distance it travels in $\rho(w)$. The chronicle terminates once the electron reaches a "new" vertex in the tree.

As an example, consider a wire w in $\mathbb{R}^3$ that is incident with a Steiner point $Q = (q_1, q_2, q_3)$. Suppose w consists of $s(w) = 3$ segments, traveling (in order from Q) one unit in the positive x_2 direction, one unit in the negative x_3 direction, and two units in the positive x_1 direction, terminating at a point with coordinates (q_1+2, q_2+1, q_3-1). Then, w has wire code and distance code

$$c(w) = (\epsilon_1 i_1, \epsilon_2 i_2, \epsilon_3 i_3) = (+2, -3, +1) \tag{3.4}$$

and

$$\rho(w) = (\rho_1, \rho_2, \rho_3) = (1, 1, 2), \tag{3.5}$$

respectively.

Since the Steiner point of interest will always be made clear when using wire and distance codes, we have suppressed Q from the notation of the codes.

We will find wire codes convenient in the next section, where we define the local topology of a Steiner point.

4. The local topology of a Steiner point in $\mathbb{R}^d$. In this section, we characterize the local topology of a Steiner point and show that we can locally restructure a Steiner tree to a canonical form that will benefit us in subsequent sections.

Consider first a Steiner point Q with degree $d(Q)$. If we arbitrarily label the wires incident with Q as $w_1, w_2, \ldots, w_{d(Q)}$, then we can completely specify the *local topology* of the Steiner point Q as wire codes $c(w_1), c(w_2), \ldots, c(w_{d(Q)})$ and distance codes $\rho(w_1), \rho(w_2), \ldots, \rho(w_{d(Q)})$, bearing in mind that each code $c(w_j)$ or $\rho(w_j)$ is an $s(w_j)$-tuple. The components of the $c(w_j)$ are symbol pairs, and the components of the $\rho(w_j)$ are Euclidean lengths. We have the following lemma.

LEMMA 4.1. *If Q is a Steiner point of an* MRST, *then, for any distinct wires w and w' incident with Q, the wire codes $c(w)$ and $c(w')$ can share no symbol pair.*

Proof. Let T be an MRST of a point set S and let Q be a Steiner point of T. We proceed by contradiction. Without loss, assume that the symbol pair ϵi appears in both $c(w)$ and $c(w')$. We will reconstruct the local topology of Q to produce a tree T' such that $\sum_{w \in T'} r(w) < \sum_{w \in T} r(w)$, where $r(w)$ is the rectilinear length of w; this will contradict the minimality of T.

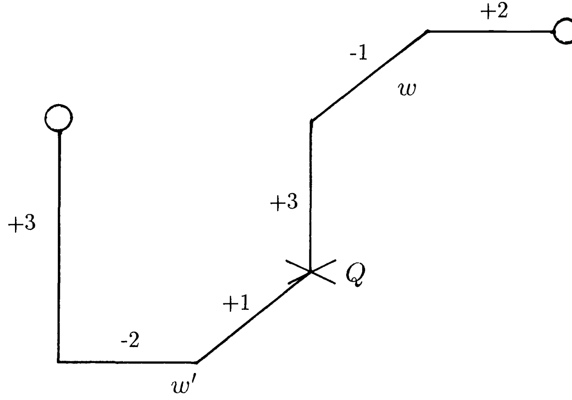


FIG. 1. *Wires w and w', whose wire codes share the symbol pair ϵi in dimension* 3. *Wire codes appear adjacent to their corresponding segments, Q is a Steiner point, and $\epsilon i = +3$.*

The initial situation is reflected in the following codes:

$$(4.1)\quad \begin{aligned} c(w) &= (\epsilon_1 i_1, \epsilon_2 i_2, \ldots, \epsilon_{j-1} i_{j-1}, \epsilon i, \epsilon_{j+1} i_{j+1}, \ldots, \epsilon_{s(w)} i_{s(w)}), \\ \rho(w) &= (\rho_1, \rho_2, \ldots, \rho_{j-1}, \rho_j, \rho_{j+1}, \ldots, \rho_{s(w)}); \\ c(w') &= (\epsilon'_1 i'_1, \epsilon'_2 i'_2, \ldots, \epsilon'_{k-1} i'_{k-1}, \epsilon i, \epsilon'_{k+1} i'_{k+1}, \ldots, \epsilon'_{s(w')} i'_{s(w')}), \\ \rho(w') &= (\rho'_1, \rho'_2, \ldots, \rho'_{k-1}, \rho'_k, \rho'_{k+1}, \ldots, \rho'_{s(w')}). \end{aligned}$$

This notation uses primes to denote the components of the codes for w', and the common symbol pair ϵi appears as the jth and kth components of $c(w)$ and $c(w')$, respectively.

We can replace w and w' with a set of three wires, $\bar{w}^\star$, $w^\star$, and $w'^\star$, as shown in Figs. 1 and 2, to form the tree T'. The wire $\bar{w}^\star$ is a single-segment wire that travels in the ϵi direction from Q to a new Steiner point $Q^\star$, which is located at a distance of $\min\{\rho_j, \rho'_k\}$ from Q.

Without loss, assume that $\rho_j \le \rho'_k$. The wire $w^\star$ is then defined to be w with its jth segment removed, and $w^\star$ originates from the new Steiner point $Q^\star$. The wire $w'^\star$ is identical to w', except that we replace the kth distance code component ρ'_k with $\rho'_k - \rho_j$ and again originate from $Q^\star$. Figures 1 and 2 demonstrate that these operations really just combine the segments of w and w' that have the same symbol pair ϵi into a single wire, $\bar{w}^\star$, which travels to a new Steiner point. This allows us to remove a segment of w or w'. (In this case, where $\rho_j \le \rho'_k$, we remove a segment from w.) Formally, the new set of wire and distance codes (assuming $\rho_j \le \rho'_k$) is

$$(4.2)\quad \begin{aligned} c(\bar{w}^\star) &= (\epsilon i), \\ \rho(\bar{w}^\star) &= (\min\{\rho_j, \rho'_k\}) = (\rho_j); \\ c(w^\star) &= (\epsilon_1 i_1, \epsilon_2 i_2, \ldots, \epsilon_{j-1} i_{j-1}, \epsilon_{j+1} i_{j+1}, \ldots, \epsilon_{s(w^\star)} i_{s(w^\star)}), \\ \rho(w^\star) &= (\rho_1, \rho_2, \ldots, \rho_{j-1}, \rho_{j+1}, \ldots, \rho_{s(w^\star)}); \\ c(w'^\star) &= (\epsilon'_1 i'_1, \epsilon'_2 i'_2, \ldots, \epsilon'_{k-1} i'_{k-1}, \epsilon i, \epsilon'_{k+1} i'_{k+1}, \ldots, \epsilon'_{s(w'^\star)} i'_{s(w'^\star)}), \\ \rho(w'^\star) &= (\rho'_1, \rho'_2, \ldots, \rho'_{k-1}, \rho'_k - \rho_j, \rho'_{k+1}, \ldots, \rho'_{s(w'^\star)}). \end{aligned}$$

An easy calculation shows that, proceeding from the new Steiner point $Q^\star$, the wires $w^\star$ and $w'^\star$ terminate in the same places as do w_1 and w_2, respectively. Hence, T' is a Steiner tree of S.

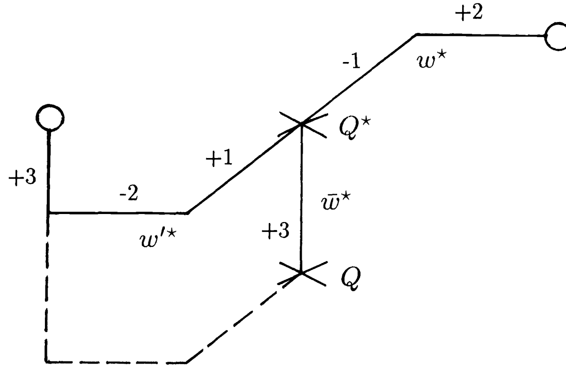


FIG. 2. *The wires $\bar{w}^\star$, $w^\star$, and $w'^\star$, which replace the suboptimal wires w and w' of Fig. 1. (The dotted line indicates parts of w' that are no longer present.) $Q^\star$ is the new Steiner point.*

Let Δ be the difference in total weight between the trees T' and T. Then,

$$
\begin{aligned}
\Delta &= \sum_{w \in T'} r(w) - \sum_{w \in T} r(w) \\
&= r(\bar{w}^\star) + r(w^\star) + r(w'^\star) - (r(w_1) + r(w_2)) \qquad (4.3) \\
&= \rho_j + (\rho'_k - \rho_j) - (\rho_j + \rho'_k) \\
&< 0,
\end{aligned}
$$

so $\sum_{w \in T'} r(w) < \sum_{w \in T} r(w)$. This contradicts the minimality of the MRST T and proves the lemma. □

Let $\epsilon_j i_j$ and $\epsilon_k i_k$ be two symbol pairs appearing somewhere in the wire codes of the local topology of a Steiner point Q, though not necessarily in the same wire code. If $\epsilon_j \neq \epsilon_k$ or if $i_j \neq i_k$, then we say the symbol pairs $\epsilon_j i_j$ and $\epsilon_k i_k$ are *distinct*. When we originate from a Steiner point and traverse a wire or wires, distinct symbol pairs correspond to segments that either parallel different coordinate axes or parallel the same axis but travel in opposite directions.

If all the symbol pairs of the local topology of Q are distinct, and if, for each wire code $c(w)$ in the local topology of Q, the dimensional symbols i_j in $c(w)$ are distinct, then we say the local topology is in *canonical form*. By Lemma 4.1, we see that if T is an MRST, then the only place a symbol pair can appear twice in a local topology is within the same wire code. This situation is handled by the following lemma.

LEMMA 4.2. *Any wire incident with a Steiner point of a Steiner tree T can be removed and replaced by a wire that contains distinct dimensional symbols i_j, if necessary, without increasing the total weight of T.*

Proof. Let w be a wire incident with a Steiner point Q of T. If w contains no multiple occurrences of a symbol, then there is no need to modify T.

Suppose therefore that the local topology of Q is not in canonical form; i.e., suppose there are two or more identical symbols in w. In this case, it is easily seen that w can be replaced with a wire with its segments reordered so that identical symbols appear contiguously. These segments can then be combined as one segment. Doing this iteratively forms a wire with at most d segments, all of whose dimensional symbols are unique. □

Lemmas 4.1 and 4.2 tell us that we can easily change the local topology of any Steiner point to obtain a topology in canonical form, without altering the total weight of the tree. This gives us the following.

COROLLARY. *For any set of point $S \subset I\!R^d$, there exists an* MRST *T of S such that if Q is a Steiner point of T, then the local topology of Q is in canonical form.*

With this corollary, we can now better characterize the local topology of a Steiner point; this is done in the next section.

Before doing so, we note for later reference that a canonical form of a local topology is not unique, for we can arbitrarily permute the components of any wire code of an MRST T (and their respective components in the corresponding distance code) to obtain a tree T' such that $\sum_{w\in T'} r(w) = \sum_{w\in T} r(w)$. Since T and T' are also isomorphic, we say that T and T' are *equivalent*.

5. The view from a Steiner point. To complete the proof of the Generalized Hanan Theorem, we must prove that, given a point set $S \subset I\!R^d$, there exists an MRST T of S such that all Steiner points of T lie at intersection points of $H(S)$, the Hanan Grid formed by S. We can "view" the theorem from the perspective of a Steiner point of T, as in the following equivalent formulation.

THEOREM (ALTERNATE GENERALIZED HANAN THEOREM). *There exists an* MRST *T of $S \subset I\!R^d$ such that, if Q is a Steiner point of T, then, for all $1 \leq i \leq d$,*

$$N(Q,i) \text{ contains a point } P \in S. \tag{5.1}$$

This version of the theorem says that in each of the d hyperplanes containing the Steiner point Q and normal to the d coordinate axes, we will find a member of the original point set S. If this is the case, then Q must be an intersection point of the Hanan Grid $H(S)$.

The goal of this section is to prove a relaxed version of the Alternate Generalized Hanan Theorem. We first look at what happens as we traverse a wire w incident with a Steiner point Q, whose local topology is in canonical form. Beginning at Q, the origin Q itself is contained in all hyperplanes $N(Q,i)$. If we next travel one segment, say, in a direction parallel to the x_j-axis, then we arrive at a vertex that is contained in all the $N(Q,i)$ except for $N(Q,j)$. If we now travel one segment parallel to the x_k-axis, then we arrive at a vertex contained in all the $N(Q,i)$ but $N(Q,j)$ and $N(Q,k)$. Because the local topology of Q is in canonical form, w can never "return" to the hyperplanes $N(Q,j)$ and $N(Q,k)$ once it has traveled in the jth and kth directions. This generalizes to the following lemma.

LEMMA 5.1. *Let w be a wire incident with a Steiner point Q whose local topology is in canonical form. If*

$$c(w) = (\epsilon_1 i_1, \epsilon_2 i_2, \ldots, \epsilon_{s(w)} i_{s(w)}) \tag{5.2}$$

and we traverse w from Q, then w must terminate at a vertex P such that

$$P \in \bigcap_{i\in I} N(Q,i), \tag{5.3}$$

where $I = \{1, 2, \ldots, d\} - \{i_1, i_2, \ldots, i_{s(w)}\}$; i.e.,

$$P \notin \{N(Q,i_1), N(Q,i_2), \ldots, N(Q,i_{s(w)})\}. \tag{5.4}$$

□

So, if $s(w) = k$, then w terminates at a vertex that lies in all but k of the $N(Q,i)$. (Note that if $s(w) = d$, then w terminates at a vertex that lies in none of the $N(Q,i)$.) This is summarized in the following.

COROLLARY. *Let Q be a Steiner point whose local topology is in canonical form. If w is a wire incident with Q and terminating at a vertex P, then*

$$\left|\{\, i : P \in N(Q,i)\,\}\right| = d - s(w). \tag{5.5}$$ □

Let $\eta(Q) = \{\, P : P \text{ is a vertex adjacent to } Q \text{ in } T\,\}$; this is the set of neighbors of Q. The next lemma tells us that, for any given point set S, there exists an MRST T of S such that if Q is a Steiner point of T, then each $N(Q,i)$, where $1 \le i \le d$, contains a neighbor of Q. In many ways, this captures much of the essence of the Generalized Hanan Theorem.

LEMMA 5.2. *There exists an* MRST *T of $S \subset \mathbb{R}^d$ such that, if Q is a Steiner point of T, then, for each $1 \le i \le d$, there exists a vertex P and a wire w connecting Q to P such that*

$$\begin{aligned} &\text{(i) } P \in N(Q,i) \textstyle\bigcap \eta(Q) \text{ and} \\ &\text{(ii) } w \text{ is entirely contained in } N(Q,i). \end{aligned} \tag{5.6}$$

Proof. Let T be an MRST of S such that if Q is a Steiner point of T, then the local topology of Q is in canonical form. The existence of such an MRST is guaranteed by the corollary to Lemma 4.2.

Consider a vertex P adjacent to Q via wire w. The $N(Q,i)$ to which P does not belong are determined by the wire code $c(w)$ as specified be Lemma 5.1. We note that the ϵ_j are irrelevant in determining the $N(Q,i)$ to which P belongs, for the ϵ_j are only directional parameters.

In eliminating any $N(Q,k)$ from the list of normal hyperplanes containing P, $c(w)$ must contain the symbol pair $\epsilon_i k$ for some $1 \le i \le s(w)$. But, since the local topology of Q is in canonical form, its set of wire codes consists, by definition, of distinct symbol pairs. This means that the symbol k can appear in a symbol pair at most twice, once as $+k$ and once as $-k$ in the wire codes of the local topology of Q. This eliminates at most two members of $\eta(Q)$ from inclusion in the normal hyperplane $N(Q,k)$, as well as at most two wires incident with Q.

By Lemma 3.1(i), there must therefore be at least one wire incident with Q that terminates at a vertex belonging to $N(Q,k)$. Since this is true for all $1 \le k \le d$, the lemma is proved. □

We note that Lemma 5.2 is similar to the alternate version of the Generalized Hanan Theorem, except that we have dropped the requirement that the neighbor $P \in S$. So, since in each hyperplane $N(Q,i)$, where $1 \le i \le d$, we find a member of $\eta(Q)$, if none of the neighbors of Q are Steiner points, then we have proved the theorem. If, however, Q is adjacent to at least one Steiner point Q', then we still have to prove that each $N(Q,i)$ to which Q' belongs contains a member of S. In the next section we introduce trombone wires, which will allow us to prove that we can find an MRST with this property.

6. Trombone wires. In the last section, we saw that any Steiner point Q with canonical local topology has a member of its neighbor set $\eta(Q)$ in each of the normal hyperplanes $N(Q,i)$. If $\eta(Q) \subset S$, then Q satisfies the requirements of the Alternate Generalized Hanan Theorem. But, unfortunately, $\eta(Q)$ may contain Steiner points, and, worse, we will soon see that some of the wires of an MRST can occupy an infinite number of locations.

Consider the Steiner points Q and Q', joined by the wire w in Fig. 3. Since the sole purpose of Q, Q', and w is to join the segments AB and CD, w is free to "slide"

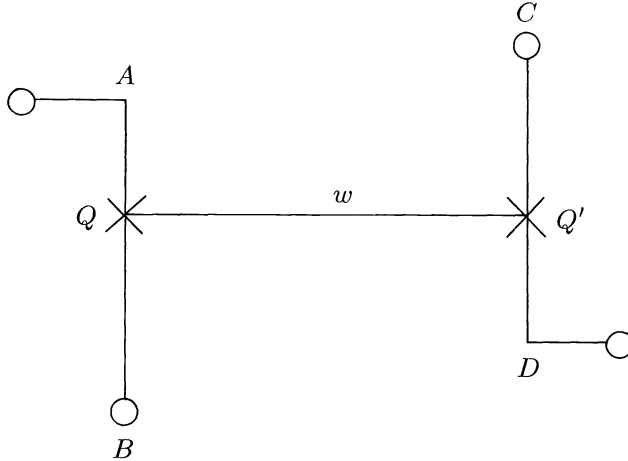


FIG. 3. *A trombone wire w in $d = 2$. Q and Q' are Steiner points.*

into positions parallel to itself along AB and CD without changing the total weight of the MRST. This means that, in general, Q and Q' can occupy an infinite number of locations and need not belong to the set of intersection points of the Hanan Grid $I_{H(S)}$. To prove the Generalized Hanan Theorem, we must find places for wires like w that guarantee containment in $I_{H(S)}$ for all Steiner points.

Because w is free to travel along the segments AB and CD in Fig. 3, we call w a *trombone wire.* The construction shown in Fig. 3 is a special case of the following, more general, definition, which is illustrated by Fig. 4. Let Q be a Steiner point of an MRST in general dimension (with all local topologies in canonical form), such that the local topology of Q contains the symbol pairs $+i$ and $-i$. Let Q' be a Steiner point joined to Q by a wire w so that $Q' \in N(Q,i)$. Note that the Steiner point Q' is guaranteed to exist by Lemma 5.2 if $N(Q,i) \cap I_{H(S)} = \emptyset$, i.e., if the hyperplane $N(Q,i)$ contains no intersection point of the Hanan Grid. Since $Q' \in N(Q,i)$, w cannot contain the symbol i, i.e., w is entirely contained in $N(Q,i)$.

More generally, consider the (connected) subtree τ containing Q and Q' that lies entirely in $N(Q,i)$. None of the wire codes corresponding to wires in τ can contain the symbol i. On the other hand, the wire codes of the local topologies of the Steiner points of τ may contain $+i$ and/or $-i$.

For all Steiner points Q' of τ (including Q) that are incident with wires whose codes contain the symbol i, if the symbol pair containing i always appears as the first symbol in its wire code, then we call the wires of τ *trombone wires*, and τ a *trombone tree.* Figure 4 shows a trombone tree in dimension three; note that τ and all its trombone wires are entirely contained in $N(Q,i)$. Note also that if $N(Q,i) \cap I_{H(S)} = \emptyset$, then all the vertices of τ must be Steiner points.

We now set out to prove various properties, including existence, of trombone trees. Note first that, by definition, trombone trees contain no less than two Steiner points.

Our main concern is with Steiner points that do not belong to $I_{H(S)}$. Let T be an MRST such that all the local topologies of T are in canonical form, and let

$$\psi(T) = \{\, Q : Q \text{ is a Steiner point of } T \text{ and } Q \neq I_{H(S)} \,\}. \tag{6.1}$$

In other words, $\psi(T)$ is the set of Steiner points of T that do not lie at intersection points of $H(S)$. The following lemma tells us that if there is at least one Steiner point

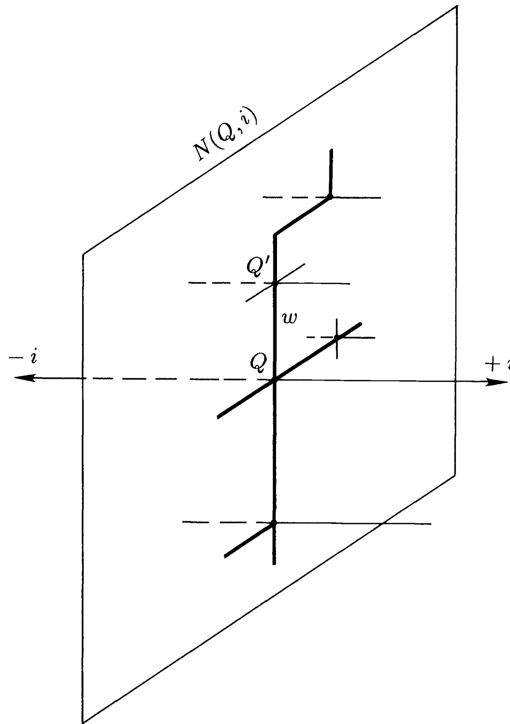


FIG. 4. *Part of a trombone tree in* $d = 3$. *The directions* $+i$ *and* $-i$ *are indicated, and the plane is* $N(Q, i)$. *The trombone wires are darkened.*

in T not coincident with an intersection point of $H(S)$, then there exists a member of $\psi(T)$ that is adjacent to exactly three vertices, two of which are points of $I_{H(S)}$.

LEMMA 6.1. *If* T *is an* MRST *of* $S \subset \mathbb{R}^d$ *such that the local topologies of all Steiner points of* T *are in canonical form, and if* $\psi(T) \neq \emptyset$, *then there exists a Steiner point* $Q \in \psi(T)$ *such that*

$$\text{(6.2)} \qquad \begin{array}{l} \text{(i) } d(Q) = 3 \textit{ and} \\ \text{(ii) } |\eta(Q) \bigcap I_{H(S)}| = 2. \end{array}$$

Proof. Consider the subgraph G generated by $\psi(T)$. Since, by Lemma 3.1, $d(Q) \geq 3$ for all $Q \in \psi(T)$, each leaf of G must be adjacent in T to at least two members of $I_{H(S)}$. Since $\psi(T) \neq \emptyset$, G must contain at least one leaf.

To prove the equalities of (6.2), let Q be a leaf of G. Since $Q \in \psi(T)$, there is a dimension x_i such that $N(Q, i) \bigcap I_{H(S)} = \emptyset$. The members of $I_{H(S)}$ connected to Q, then, must not belong to $N(Q, i)$. This means that the wire codes of the wires connecting Q to the points of $I_{H(S)}$ both contain the symbol i. Since the local topology of Q is in canonical form, there can be at most two such wires; this proves property (ii). All remaining wires incident with Q must be contained entirely in $N(Q, i)$.

But, since all the neighbors $\eta(Q)\bigcap N(Q,i)$ must belong to G and since Q is a leaf, $d(Q)=3$. □

We remark that the proof of property (ii) in Lemma 6.1 is considerably simpler than its dimension-two counterpart in Hanan (1966).

We need one more lemma before discussing the existence of trombone wires. Consider $Q \in \psi(T)$, and let x_i be such that $N(Q,i)\bigcap I_{H(S)} = \emptyset$. By Lemma 5.2, there exists at least one point $Q' \in \eta(Q)$ such that $Q' \in N(Q,i)$. Consider Q'. Since $N(Q,i)\bigcap S = \emptyset$, Q' must be a Steiner point. Moreover, $Q' \notin I_{H(S)}$, so $Q' \in \psi(T)$. We summarize this in the following lemma.

LEMMA 6.2. *If T is an* MRST *of $S \subset \mathbb{R}^d$ such that the local topologies of all Steiner points of T are in canonical form, and if $Q \in \psi(T)$, then there exist a Steiner point Q' of T and a dimension x_i such that*

$$\text{(6.3)}\qquad \begin{aligned} &\text{(i) } N(Q,i)\bigcap I_{H(S)} = \emptyset;\\ &\text{(ii) } Q' \in \eta(Q);\\ &\text{(iii) } Q' \in N(Q,i); \text{ and}\\ &\text{(iv) } Q' \in \psi(T). \end{aligned}$$

Moreover,

$$\text{(6.4)}\qquad \{\, Q' : Q' \in \eta(Q)\bigcap N(Q,i)\,\} \subset \psi(T). \qquad □$$

The relation (6.4) means that property (6.3(iv)) is satisfied by *all* (Steiner) points Q' adjacent to Q that lie in $N(Q,i)$.

We use Lemmas 6.1 and 6.2 to prove the following key lemma, which sets the stage for the movement of trombone wires in the next section. It essentially tells us that if $\psi(T) \neq \emptyset$, then we can find a trombone tree "rooted" at some Steiner point adjacent to two members of $I_{H(S)}$.

LEMMA 6.3. *If T is an* MRST *of $S \subset \mathbb{R}^d$ such that the local topologies of all Steiner points of T are in canonical form, and if $\psi(T) \neq \emptyset$, then there exists a tree T' equivalent to T, a Steiner point $Q \in \psi(T')$, a dimension x_i, and a subgraph τ of T' such that*

$$\text{(6.5)}\qquad \begin{aligned} &\text{(i) } |\eta(Q)\bigcap I_{H(S)}| = 2;\\ &\text{(ii) } d(Q) = 3;\\ &\text{(iii) } Q \in \tau; \text{ and}\\ &\text{(iv) } \tau \text{ is a trombone tree contained entirely in } N(Q,i). \end{aligned}$$

Proof. We first remark that for any tree T' equivalent to T, $\psi(T) = \psi(T')$ by the definition of equivalent trees (at the end of §4), since T and T' share identical vertex sets.

Let Q be a member of $\psi(T)$ attached to two members of $I_{H(S)}$ and one member of $\psi(T)$, as guaranteed by Lemma 6.1, and let i be such that $N(Q,i)\bigcap I_{H(S)} = \emptyset$, as guaranteed by Lemma 6.2.

Since the local topology of Q is in canonical form, the two wires incident with Q that terminate at points in $I_{H(S)}$ are precisely the wires in the local topology of Q that contain the symbol pairs $+i$ and $-i$; the other wire incident with Q is contained entirely in $N(Q,i)$.

By Lemma 6.2, there must be at least one Steiner point $Q' \in \eta(Q)\bigcap N(Q,i)$. To form T' from T, consider the local topologies of all $Q' \in N(Q,i)$, including Q, and,

in every wire code $c(w)$ containing the symbol i, rearrange the symbol pairs of $c(w)$ (and the corresponding components of the distance code $\rho(w)$) so that the symbol pair containing i appears first.

It is easily seen that the connected subgraph of T' containing Q and lying entirely in $N(Q,i)$ is a trombone tree. □

Since we will eventually want to move the trombone wires of an MRST so that all its Steiner points are members of $I_{H(S)}$, we need to know when, where, and how far we can move the trombone trees of an MRST. This issue is handled in the next section.

7. Trombone lemmas. Using the lemmas of the preceding section, we can now consider sliding the trombone wires of an MRST T into place. In the last section, it was proved that if $\psi(T) \neq \emptyset$, then we can find a certain trombone tree in T. From the definition of a trombone tree, however, it is not clear that we are able to move the tree without sacrificing T's minimality.

Let T be an MRST such that $\psi(T) \neq \emptyset$ and all local topologies of T are in canonical form, and let τ be a trombone tree in T containing $Q \in \psi(T)$ satisfying properties (i) through (iv) of Lemma 6.3. Since none of the vertices of τ belong to $I_{H(S)}$, Lemma 6.3 would seem to imply that τ is free to move in the $+i$ or $-i$ direction for some positive distance.

If, however, there exists a Steiner point Q' in τ such that the local topology of Q' contains a wire w^- whose wire code $c(w^-)$ begins with $-i$, yet no wire code in the local topology of Q' begins with $+i$, then sliding τ in the $+i$ direction may increase the total weight of the MRST. This is true because if Q' is to remain adjacent to the point to which it is connected by w^-, then moving τ in the $+i$ direction must increase the length of the first segment of w^-, which we denote by $\rho_1(w^-)$.

The next lemma guarantees us that for every wire such as w^-, there is another wire w^+ such that $c(w^+)$ begins with the symbol pair $+i$. Let $c_1(w)$ be the first symbol pair of the wire code $c(w)$, and let

$$\begin{aligned} W^+(\tau,i) &= \{\, w : c_1(w) = +i \text{ and } c(w) \text{ is in the local topology of} \\ &\qquad\qquad \text{a Steiner point of } \tau \,\} \text{ and} \\ W^-(\tau,i) &= \{\, w : c_1(w) = -i \text{ and } c(w) \text{ is in the local topology of} \\ &\qquad\qquad \text{a Steiner point of } \tau \,\}. \end{aligned} \tag{7.1}$$

In words, $W^+(\tau,i)$ and $W^-(\tau,i)$ are the sets of wires emanating from the trombone tree τ in the $+i$ and $-i$ directions, respectively. These wires are orthogonal to the hyperplane containing τ.

LEMMA 7.1. *If τ is a trombone tree in T, an* MRST *of $S \subset I\!R^d$ whose local topologies are in canonical form, and if τ is entirely contained in $N(Q,i)$, where $N(Q,i) \bigcap I_{H(S)} = \emptyset$, then*

$$|W^+(\tau,i)| = |W^-(\tau,i)|. \tag{7.2}$$

Proof. Without loss, suppose that $|W^+(\tau,i)| > |W^-(\tau,i)|$. If we move τ in the $+i$ direction any distance $\delta > 0$ such that all wire codes of the local topologies of T are preserved, then the first segments of the wires in $W^-(\tau,i)$ will increase in length by δ, while the first segments of the wires in $W^+(\tau,i)$ will decrease by δ. If we let $\overline{T}$ be the tree after moving τ, then

$$\sum_{w\in\overline{T}} r(w) - \sum_{w\in T} r(w) = \delta\,|W^-(\tau,i)| - \delta\,|W^+(\tau,i)| < 0, \tag{7.3}$$

so $\sum_{w\in\overline{T}} r(w) < \sum_{w\in T} r(w)$. But, since all local topologies of T were preserved when moving τ, $\overline{T}$ is a rectilinear Steiner tree of S. This contradicts the minimality of T, proving the lemma. □

Lemma 7.1 allows us to move a trombone tree τ contained in some $N(Q,i)$ in either the $+i$ or $-i$ direction. The next lemma tells us *how far* we can slide τ while preserving the minimality of the MRST to which τ belongs. Recall that $\rho_1(w)$ is the first entry in the distance code of w, i.e., the length of the first segment of w.

LEMMA 7.2. *Let T be an* MRST *of $S \subset \mathbb{R}^d$ whose local topologies are in canonical form. If τ is a trombone tree in T such that τ is entirely contained in $N(Q,i)$, where $N(Q,i)\bigcap I_{H(S)} = \emptyset$, then τ can be moved a distance of δ^+ and δ^- in the $+i$ and $-i$ directions, respectively, where*

$$0 < \delta^+ \leq \min\{\,\rho_1(w) : w \in W^+(\tau,i)\,\} \quad \text{and} \qquad (7.4)$$
$$0 < \delta^- \leq \min\{\,\rho_1(w) : w \in W^-(\tau,i)\,\},$$

to obtain a tree $\overline{T}$ such that $\sum_{w\in\overline{T}} r(w) = \sum_{w\in T} r(w)$.

Proof. If we slide τ in the $+i$ direction a distance of $\delta^+ < \min\{\,\rho_1(w) : w \in W^+(\tau,i)\,\}$, then none of the wire codes of the local topologies of T can change, for all vertices of τ belong to $\psi(T)$, and we have not moved τ far enough to eliminate any symbol pairs from the wire codes. By Lemma 7.1, $|W^+(\tau,i)| = |W^-(\tau,i)|$, so, by moving τ, we have increased and decreased an equal number of distance code components, preserving the total weight of T. Note that as τ is moved its wires do not intersect wires contained in a hyperplane parallel to $N(Q,i)$ to create a cycle. This would contradict the minimality of T.

Suppose now that $\delta^+ = \min\{\,\rho_1(w) : w \in W^+(\tau,i)\,\}$. In this case, we have moved τ such that the local topology of at least one Steiner point Q of τ no longer has $+i$ in a wire code that contained $+i$ before τ was slid. (In fact, if the wire code originally consisted of only one component, then the wire has disappeared altogether.) It can be shown in this case that the resulting tree $\overline{T}$ is still an MRST of S, and, using the same logic applied in the $\delta^+ < \min\{\,\rho_1(w) : w \in W^+(\tau,i)\,\}$ case, $\sum_{w\in\overline{T}} r(w) = \sum_{w\in T} r(w)$.

The proof for δ^- is analogous. □

All that remains is to move the trombone wires of T' into position such that all Steiner points coincide with points of $I_{H(S)}$. This is done in the next section, where we conclude the proof of the Generalized Hanan Theorem.

8. Sliding the trombone wires. In this section, we conclude the proof of the Generalized Hanan Theorem by sliding trombone wires into place so that all Steiner points of an MRST T of $S \subset \mathbb{R}^d$ belong to $I_{H(S)}$.

We first ensure that the local topologies of T are in canonical form. Then, if $\psi(T) \neq \emptyset$, we consider $Q \in \psi(T)$, the equivalent MRST T', and the trombone tree τ that satisfy properties (i) through (iv) of Lemma 6.3.

For every point $Q \in \psi(T')$, there is a set of dimensions

$$v(Q,T') = \{\,i : N(Q,i)\bigcap I_{H(S)} = \emptyset\,\}; \qquad (8.1)$$

we call $v(Q,T')$ the *violation set* of Q in T', for it is a list of all dimensions whose normal hyperplanes containing Q are "in violation" of the (Alternate) Generalized Hanan Theorem. The goal is to make $v(Q,T') = \emptyset$ for all Steiner points Q in T'.

The following lemma tells us that we can slide the trombone tree τ so that a dimension i is removed from $v(Q,T')$ without simultaneously adding any dimensions

to $v(Q, T')$ or any points to $\psi(T')$. The resulting tree $\overline{T}$, if $v(Q, \overline{T})$ is not yet empty, still satisfies the conditions required by Lemma 6.3, so we still have a Steiner point, a dimension $j \neq i$, and a trombone tree in $\overline{T}$ satisfying properties (i) through (iv) in Lemma 6.3. This allows us to again apply Lemma 8.1, and we can do so until $Q \in I_{H(S)}$. Hence, Lemma 8.1 suffices to prove the Generalized Hanan Theorem: we simply slide trombone wires in T' until $\psi(T')$ is empty.

LEMMA 8.1. *Let Q be a Steiner point of T, an* MRST *of $S \subset \mathbb{R}^d$ such that all local topologies of T are in canonical form, with $|\eta(Q) \bigcap I_{H(S)}| = 2$ and $d(Q) = 3$. If $Q \in \psi(T)$ and if τ is a trombone tree contained entirely in $N(Q, i)$, where $N(Q, i) \bigcap I_{H(S)} = \emptyset$, then τ can be moved in the $+i$ or $-i$ direction to a new location to obtain a tree $\overline{T}$ such that*

$$\begin{aligned} &\text{(i) } \overline{T} \textit{ is an MRST of } S; \\ &\text{(ii) } \textit{all local topologies of } \overline{T} \textit{ are in canonical form;} \\ &\text{(iii) } |v(Q, \overline{T})| \leq |v(Q, T)| - 1; \textit{ and} \\ &\text{(iv) } |\psi(\overline{T})| \leq |\psi(T)|. \end{aligned} \tag{8.2}$$

Proof. Note first that neither point of $I_{H(S)}$ attached to Q can belong to $N(Q, i)$, so the wire codes of the local topology of Q corresponding to wires terminating at members of $I_{H(S)}$ must, respectively, contain the symbol pairs $+i$ and $-i$. Let these wires be w^+ and w^-, respectively.

Move τ a distance of $\delta^+ = \min\{\, \rho_1(w) : w \in W^+(\tau, i) \,\}$ in the $+i$ direction. Note that δ^+ is within the bounds required by Lemma 7.2, so the resulting tree is still an MRST, and note also that no symbol pairs have been added to any wire codes and no existing symbol pairs have been altered, so all local topologies are still in canonical form.

Let Q' be a Steiner point in τ. As τ is moved, every wire incident with Q' that lies entirely in $N(Q, i)$ is moved. There are at most two wires incident with Q' that are not entirely contained in $N(Q, i)$ (and Q' slides along these wires, if they exist). Therefore, if we consider as a vertex the location occupied by Q' before sliding τ, then the degree of the vertex after moving τ is less than three. This means that when we slide τ, there is no need to "split" Q' into two Steiner points, one at its old location, and one at its new location. In other words, $|\psi(T)|$ does not increase as we slide τ.

The key issue, then, is property (iii). As a first case, if $\delta^+ = \rho_1(w^+)$, then we claim the lemma is proved. To prove the claim, note that w^+ joins Q to a member of $I_{H(S)}$, and let Q' be the location of Q after sliding τ. Since the local topology of Q in T is in canonical form, the remainder of w^+, including its terminal member of $I_{H(S)}$, must be in $N(Q', i)$. This removes i from $v(Q, \tau)$, and since, for all j, where $1 \leq j \leq d$ and $j \neq i$, $N(Q, j)$ does not change as τ is slid, no new dimensions are added to $v(Q, \tau)$. This proves the claim.

Suppose now that $\delta^+ \neq \rho_1(w^+)$. Let Q' be a Steiner point in τ such that a wire $w'^+ \in W^+(\tau, i)$ incident with Q' satisfies $\rho_1(w'^+) = \delta^+$, and let P be the location of Q' after sliding τ. Note that P can be a vertex of T or a "corner point" in w^+, i.e., a location at which w^+ simply changes direction and continues parallel to a different axis. If $P \in I_{H(S)}$, or, more generally, if $N(P, i) \bigcap I_{H(S)} \neq \emptyset$, then by the same reasoning as above, the lemma is proved, for Q belongs to $N(P, i)$, removing i from $v(Q, \tau)$.

So, assume as a final case that $N(P, i) \bigcap I_{H(S)} = \emptyset$ and consider the status of T_1, which we define to be the original tree T after sliding τ a distance of δ^+. The

local topologies of T_1 are still in canonical form. Since $i \in v(Q,\tau)$, $\psi(T_1)$ is not empty. Furthermore, τ is still contained in $N(Q,i)$, and $N(Q,i) \bigcap I_{H(S)} = \emptyset$, for $N(Q,i) = N(P,i)$. By Lemma 7.2, we can therefore again slide τ in the $+i$ direction, again letting $\delta^+ = \min\{\rho_1(w) : w \in W^+(\tau,i)\}$. If we continue to slide τ in this manner, τ must eventually revert to one of the previously considered cases and come to rest in a hyperplane $N(Q,i)$ such that $N(Q,i) \bigcap I_{H(S)} \neq \emptyset$. This proves the lemma, and the Generalized Hanan Theorem. □

9. Concluding remarks. The Generalized Hanan Theorem realizes new solutions to the Rectilinear Steiner Problem in general dimension. Our algorithms proceed along two directions. One uses the Generalized Hanan Theorem to reduce the Rectilinear Steiner Problem to a graph-based problem that is amenable to the Dreyfus and Wagner (1972) algorithm. This yields the fastest known running times in all dimensions. A second direction is a simple and intuitive minimum spanning tree–based algorithm that gives the first polynomial-time algorithm for the k-Steiner problem in all dimensions for $k = O(1)$.

An issue that must be addressed is that of practicality. Since the set of intersection points $I_{H(S)}$ can be as large as $|S|^d$, our algorithms still offer little hope for point sets of even modest size. Though we make no attempt here to refine the algorithm, our theorem opens the door for algorithms with better performance bounds.

The Generalized Hanan Theorem and some of the combinatorial tools used to prove it may be of use in ways other than algorithmic ones. For example, using the wire codes of local topologies in canonical form, one can easily generalize Hwang's "Corner Lemma" (Hwang (1976)), which states that, in $d = 2$, no Steiner point can be incident with more than one corner wire, where a corner wire is a wire composed of two orthogonal segments. In general dimension, Hwang's lemma becomes:

CORNER LEMMA. *If T is an* MRST *of $S \subset \mathbb{R}^d$ such that the local topologies of T are in canonical form, then the total number of corners in wires incident with any Steiner point of T is at most $2d - 3$.*

In addition to combinatorial applications such as the Corner Lemma, the Generalized Hanan Theorem also has applications in computational geometry, e.g., probabilistic and worst-case weights of MRSTs in bounded space (see Snyder (1990)), since it makes precise the locations where Steiner points can occur.

The theorem may be applicable to new heuristic algorithms for the Rectilinear Steiner Problem as well.

Acknowledgments. The author thanks Frank Hwang and Marshall Bern for providing key references and for useful comments concerning an early version of this paper. Thanks also to Raimund Seidel for pointing out an absurdity in another version.

REFERENCES

M. W. BERN, (1988), *Two probabilistic results on rectilinear Steiner trees*, Algorithmica, 3, pp. 191–204.

C. CHERNIAK, (1989). *Local network optimization in the brain*, Tech. Report, Department of Philosophy and Institute for Advanced Computer Studies, University of Maryland, College Park, MD.

F. R. K. CHUNG AND R. L. GRAHAM, (1981), *On Steiner trees for bounded point sets*, Geom. Dedicata, 11, pp. 353–361.

F. R. K. CHUNG AND F. K. HWANG, (1979), *The largest minimal rectilinear Steiner trees for a set of n points enclosed in a rectangle with given perimeter*, Networks, 9, pp. 19–36.

R. COURANT AND H. ROBBINS, (1941), *What is Mathematics?*, Oxford University Press, New York, NY.

S. E. DREYFUS AND R. A. WAGNER, (1972), *The Steiner problem in graphs*, Networks, 1, pp. 195–207.

L. R. FOULDS, (1984), *Combinatorial Optimization for Undergraduates*, Springer-Verlag, New York, NY, pp. 193–201.

L. R. FOULDS AND R. L. GRAHAM, (1982), *The Steiner problem in phylogeny is NP-complete*, Adv. Appl. Math., 3, pp. 43–49.

L. R. FOULDS, M. D. HENDY, AND D. PENNY, (1979), *A graph theoretic approach to the development of minimal phylogenetic trees*, J. Mol. Evol., 13, pp. 127–149.

M. R. GAREY, R. L. GRAHAM, AND D. S. JOHNSON, (1976), *Some NP-complete geometric problems*, in Proc. Eighth Annual ACM Symposium of Theory of Computing, New York, pp. 10–22.

M. R. GAREY AND D. S. JOHNSON, (1977), *The rectilinear Steiner tree problem is NP-complete*, SIAM J. Appl. Math., 32, pp. 826–834.

G. GEORGAKOPOULOS AND C. H. PAPADIMITRIOU, (1987), *A 1-Steiner tree problem*, J. Algorithms, 8, pp. 122–130.

E. N. GILBERT AND H. O. POLLAK, (1968), *Steiner minimal trees*, SIAM J. Appl. Math., 16, pp. 1–29.

D. GUSFIELD, (1991), *Efficient algorithms for inferring evolutionary trees*, Networks, 21, pp. 19–28.

M. HANAN, (1966), *On Steiner's problem with rectilinear distance*, SIAM J. Appl. Math., 14, pp. 255–265.

F. K. HWANG, (1976), *On Steiner minimal trees with rectilinear distance*, SIAM J. Appl. Math., 30, pp. 104–114.

Z. A. MELZAK, (1961), *On the problem of Steiner*, Canad. Math. Bull., 4, pp. 143–148.

F. S. ROBERTS, (1984), *Applied Combinatorics*, Prentice–Hall, Englewood Cliffs, NJ.

D. SANKOFF AND P. ROUSSEAU, (1975), *Locating the vertices of a Steiner tree in an arbitrary metric space*, Math. Prog., 9, pp. 240–246.

T. L. SNYDER, (1990), *On minimal rectilinear Steiner trees in all dimensions*, in Proc. Sixth Annual ACM Symposium on Computational Geometry, New York, pp. 311–320; also Discrete Comput. Geom., to appear.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 181–192, February 1992

©1992 Society for Industrial and Applied Mathematics
014

ON LEARNING RING-SUM-EXPANSIONS*

PAUL FISCHER† AND HANS ULRICH SIMON†

Abstract. The problem of learning ring-sum-expansions from examples is studied. Ring-sum-expansions (RSE) are representations of Boolean functions over the base $\{\wedge, \oplus, 1\}$, which reflect arithmetic operations in $GF(2)$. k-RSE is the class of ring-sum-expansions containing only monomials of length at most k. k-*term*-RSE is the class of ring-sum-expansions having at most k monomials. It is shown that k-RSE, $k \geq 1$, is learnable while k-*term*-RSE, $k \geq 2$, is not learnable if $RP \neq NP$. Without using a complexity-theoretical hypothesis, it is proven that k-RSE, $k \geq 1$, and k-*term*-RSE, $k \geq 2$ cannot be learned from positive (negative) examples alone. However, if the restriction that the hypothesis which is output by the learning algorithm is also a k-RSE is suspended, then k-RSE is learnable from positive (negative) examples only. Moreover, it is proved that 2-*term*-RSE is learnable by a conjunction of a 2-CNF and a 1-DNF. Finally the paper presents learning (on-line prediction) algorithms for k-RSE that are optimal with respect to the sample size (worst case mistake bound).

Key words. pac-learning, Boolean functions, Vapnik–Chervonenkis-dimension, worst case mistake bounds

AMS(MOS) subject classification. 68T05

1. Introduction and definitions. The learnability of various classes of Boolean formulas from examples has been intensively studied in the last few years, (see, e.g., Kearns, Li, Pitt, and Valiant, for a survey [KLPV 87]). In this paper, we study a normal form of Boolean functions with a nice algebraic structure, the ring-sum-expansion (RSE). RSE formulas are basically polynomials over the field $GF(2)$. Any Boolean function can be uniquely represented in this way, although succinct representations are only obtained for parity-like functions. We shall see that linear algebra is very helpful for the construction of learning algorithms for RSE.

Let $n \in \mathbb{N}$. A *concept* is a Boolean function with domain $\{0,1\}^n$. The set $f^{-1}(1) := \{v \in \{0,1\}^n \mid f(v) = 1\}$ is the set of *positive examples for* f while $f^{-1}(0)$ is the set of *negative examples for* f. A *sample for* f is a collection of labeled examples for f possibly with repetitions: $((v_1, \alpha_1), \cdots, (v_m, \alpha_m))$, $(v_i, \alpha_i) \in \{0,1\}^n \times \{0,1\}$, and $f(v_i) = \alpha_i$. The *size* of the sample is m. g is *consistent with* a sample if $g(v_i) = \alpha_i$ for $1 \leq i \leq m$. We say that v *satisfies* g if $g(v) = 1$.

If $\mathcal{C}_n$ is some set of Boolean formulas on n variables, then $\mathcal{C} = \bigcup_{n=1}^{\infty} \mathcal{C}_n$ is a *class of representations.* For example, $\mathcal{C}$ might be the class of Boolean polynomials (also called DNF expressions). Given some $f \in \mathcal{C}$, then POS(f) (NEG(f)) are sources of positive (negative) examples for f. Each time POS(f) is consulted it will give a positive example. D^+, respectively, D^- are distributions on $\{0,1\}^n$ such that

$$\sum_{v \in f^{-1}(1)} D^+(v) = 1\,, \quad \sum_{v \in f^{-1}(0)} D^+(v) = 0\,, \quad \sum_{v \in f^{-1}(0)} D^-(v) = 1\,, \quad \sum_{v \in f^{-1}(1)} D^-(v) = 0\,.$$

By $\|f\|$ we denote the size of the formula f.

A *learning algorithm for* f is an algorithm that has access to POS(f) and NEG(f). A call to POS(f) or NEG(f) will count as one step. A class $\mathcal{C}$ of representations is

*Received by the editors March 23, 1990; accepted for publication (in revised form) February 25, 1991.

†Lehrstuhl Informatik II, Universität Dortmund, Postfach 500 500, D-4600 Dortmund 50, Germany.

learnable if there is a polynomial p and a learning algorithm L such that for all n, all $f \in \mathcal{C}_n$, all distributions D^+, D^- as above, and all $\varepsilon > 0$ and all $\delta > 0$,

— L halts in time $p(n, \|f\|, \frac{1}{\varepsilon}, \frac{1}{\delta})$,

— L outputs a formula $g \in \mathcal{C}_n$, called *hypothesis*, that with probability at least $(1-\delta)$ has the property

$$\sum_{v \in g^{-1}(0)} D^+(v) < \varepsilon \quad \text{and} \quad \sum_{v \in g^{-1}(1)} D^-(v) < \varepsilon.$$

A class $\mathcal{C}$ of representations is *pos-learnable* (*neg-learnable*) if $\mathcal{C}$ is learnable by a learning algorithm that uses only positive (negative) examples.

A class $\mathcal{C}$ of representations is *predictable* if there is an algorithm that meets the requirements of a learning algorithm with the exception that it may produce a hypothesis g of arbitrary type (g need not even be a Boolean formula). The terms *pos-predictable* and *neg-predictable* are defined as for learnability.

We would like to mention that what we call *learnability* is called *proper learnability* by some authors. We shall first prove some results within this framework, before we turn to the more important notion of predictability, where we allow the hypothesis to have an arbitrary syntactic form. We shall show that, with this relaxation, some problems that are not (properly) learnable are predictable.

The *consistency problem for* a class of Boolean formulas $\mathcal{C}$ is defined as follows. Given a labeled sample $((v_1, \alpha_1), \cdots, (v_m, \alpha_m))$, $(v_i, \alpha_i) \in \{0,1\}^n \times \{0,1\}$, is there a formula $f \in \mathcal{C}_n$ that is consistent with the sample.

In order to save indices we introduce the following abbreviation. For $I \subseteq \{1, \cdots, n\}$ we write x_I to denote the (monotone) monomial $\wedge_{i \in I} x_i$. We make the convention that $x_I = 1$ for $I = \emptyset$.

A *ring-sum-expansion* is a Boolean formula that is a modulo-2-sum of monomials containing only unnegated variables:

$$\bigoplus_{I \in \mathcal{A}} x_I \quad \text{for some } \mathcal{A} \subseteq \mathcal{P}(\{1, \cdots, n\})$$

where $\mathcal{P}(A)$ denotes the powerset of A. Every Boolean function can be uniquely represented as an RSE. For more details, see, e.g., Wegener [W 87].

For $k \in \mathbb{N}$, a k-RSE is a ring-sum-expansion where each monomial contains at most k variables. It will prove useful to introduce the class 1-RSE* of all 1-RSE formulas not containing the monomial 1.

For $k \in \mathbb{N}$, a k-*term*-RSE is a ring-sum-expansion consisting of at most k monomials.

We shall denote a class of representations by the type of formulas that it contains. For instance 1-RSE will denote the class of 1-RSE-formulas. Note that the learning problems for RSE, DNF, and CNF are not easily reduced to each other because the respective sizes of the formulas are not polynomially related.

2. Learnability of k-RSE. In [BEHW 87] Blumer, Ehrenfeucht, Haussler, and Warmuth gave a combinatorial criterion for learnability of Boolean formulas.

THEOREM 1 (Blumer, Ehrenfeucht, Haussler, and Warmuth). *A class $\mathcal{C}$ of representations is learnable if there are polynomials p and q such that the following conditions hold:*

(i) *For all n, $|\mathcal{C}_n| \leq 2^{p(n)}$;*

(ii) *For all r and all samples of size r a consistent representation $g \in C_n$ can be produced in time $q(r,n)$.*

THEOREM 2. 1-RSE* *is learnable.*

Proof. We shall show how to produce in polynomial time a 1-RSE* g consistent with a sample for a 1-RSE* f. Then Theorem 2 follows from Theorem 1, noting that $|\text{1-RSE}^*| = 2^n$.

Let $f \in$ 1-RSE*.

Let $S = ((v_j, \alpha_j))_{1 \le j \le m}$ be a sample for f. We set up the following linear system of equations over $GF(2)$ (the variables are the y_i):

$$y_1 v_{j1} \oplus y_2 v_{j2} \oplus \cdots \oplus y_n v_{jn} = \alpha_i \quad \text{for } 1 \le j \le m \ .$$

As f is consistent with the sample the system has a solution. It can be solved in polynomial time with respect to n and m. The set of solutions forms an affine subspace of $\{0,1\}^n$. Given any solution $(t_1, \cdots, t_n)$ define a 1-RSE* r as follows

$$r := \bigoplus_{\substack{1 \le i \le n \\ t_i = 1}} x_i.$$

Then r is consistent with the sample because

$$r(v_j) = \bigoplus_{\substack{1 \le i \le n \\ t_i = 1}} v_{ji} = \alpha_i, \qquad 1 \le j \le m \ . \qquad \square$$

Theorem 2 has been independently proved by Helmbold, Sloan and Warmuth [HSW 90]. Now the learnability of k-RSE, $k \ge 1$, follows from a substitution result in [KLPV 87a]. Moreover, the learning algorithm always produces a consistent hypothesis.

COROLLARY 3. k-RSE *is learnable,* $k \ge 1$.

3. Nonlearnability of k-term-RSE. The class 1-*term*-RSE is exactly the class of monotone monomials, which is learnable (see e.g [KLPV 87]). In [KLPV 87a] it was proved that k-*term*-DNF, $k \ge 2$, is not learnable. Here we shall prove that k-*term*-RSE, $k \ge 2$, is not learnable.

For the proof of our results we need the following theorem.

THEOREM 4. *Let C be a class of Boolean formulas. If $RP \ne NP$, then C is not learnable, if the consistency problem for C is NP-hard.*

Proof. See [KLPV 87a] for the proof. $\square$

We first turn to the nonlearnability of k-*term*-RSE. The proof is split into two parts, one for $k = 2$ and one for $k > 2$. For the case $k = 2$ we follow the proof of Kearns, Li, Pitt, and Valiant [KLPV 87a] for k-*term*-DNF. For k-*term*-RSE, $k \ge 3$, however, we need a different and more complicated proof. Observe that learnability of $(k+1)$-*term*-RSE does not imply learnability of k-*term*-RSE. Given a k-*term*-RSE r it may be possible to find a $(k+1)$-*term*-RSE r' in polynomial time that is a good approximation for r, but it can be impossible to efficiently find a k-*term*-RSE that approximates r. Thus nonlearnability of k-*term*-RSE does not imply nonlearnability of $(k+1)$-*term*-RSE.

THEOREM 5. 2-*term*-RSE *is not learnable, if* $RP \ne NP$.

Proof. The proof is similar to the one for 2-*term*-DNF in [KLPV 87a]. $\square$

We show that the consistency problem for 2-*term*-RSE is NP-hard by reducing to it the 2-colorability question for hypergraphs, which itself is NP-complete (see [GJ 79]). Theorem 5 then follows from Theorem 4.

Let an instance of the 2-colorability problem for hypergraphs be given, i.e. let $\{1,\cdots,n\}$ be the set of vertices of a hypergraph H and $S_1,\cdots,S_m \subseteq \{1,\cdots,n\}$, $|S_j| \geq 2$ be its hyperedges. Then $f:\{1,\cdots,n\} \to \{1,2\}$ is called a *2-coloring* of $\{1,\cdots,n\}$ by 1 and 2 if $|f(S_i)| = 2$ for $1 \leq i \leq m$. We define our sample by the following positive and negative examples.

— $p_i := (1,\cdots,1,0,1,\cdots,1)$, $1 \leq i \leq n$, where the only 0 is at the ith position.

— $n_i := \chi(\bar{S}_i)$, $1 \leq i \leq m$, where $\chi(\bar{S}_i)$ is the characteristic vector of S_i.

Let us first show that a 2-coloring can be converted into a consistent 2-*term*-RSE. To this end let m_j be the monomial that consists of all variables that correspond to vertices colored by j, i.e.,

$$m_j := \bigwedge_{f(i)=j} x_i, \qquad j = 1,2.$$

Let $r := m_1 \oplus m_2$. In the following we shall identify the monomials with the set of variables they contain. Thus $x_i \in m_j$ means that x_i is a variable in m_j. Note that m_1 and m_2 induce a partition of $\{1,\cdots,n\}$.

For all $1 \leq i \leq n$ the positive example p_i is evaluated to 1 by exactly one of m_1 or m_2, namely, by the unique monomial not containing x_i. Hence $r(p_i) = m_1(p_i) \oplus m_2(p_i) = 1$.

For all $1 \leq i \leq m$ the negative example n_i is evaluated to 0 by m_1 and m_2. Recall that for every hyperedge S_j example n_j has 0s at exactly those positions that correspond to vertices of S_j. As H is 2-colorable, S_i contains vertices of either color and therefore variables from both monomials. Hence $r(n_i) = m_1(n_i) \oplus m_2(n_i) = 0$.

Now let $r = m_1 \oplus m_2$ be a 2-*term*-RSE that is consistent with the examples. We show that r induces a 2-coloring of H. The variables of m_1 and m_2 form a partition of $\{1,\cdots,n\}$ because exactly one monomial maps p_i to 0 and thus exactly one monomial contains x_i. Let us define the 2-coloring f of H by

$$f(i) = j \text{ iff } x_i \in m_j, \quad 1 \leq i \leq n,\ j = 1,2\ .$$

It remains to show that no hyperedge is monochromatic. Let S_l be some hyperedge of H. Then, by consistency, $r(n_l) = m_1(n_l) \oplus m_2(n_l) = 0$, hence, $m_1(n_l) = m_2(n_l)$. Note that n_i is mapped to 0 by at least one of m_1 or m_2. Thus $m_1(n_l) = m_2(n_l) = 0$, which means that there are j, j' such that $j \in m_1$ and $j' \in m_2$ and $n_{lj} = n_{lj'} = 0$. Hence $j, j' \in S_l$ and $f(j) = 1$ and $f(j') = 2$, whence S_l is not monochromatic.

Note that the reduction is clearly polynomial. □

For the case $k \geq 3$ we need the following simple lemma.

LEMMA 6. *The problem $GC[k]$ of deciding whether a graph with maximum degree at most $2k-1$ is k-colorable is NP-complete for $k \geq 4$.*

Proof. The problem of deciding whether a graph with maximum degree 4 is 3-colorable is NP-complete (see, e.g, [GJ 79]). We describe the reduction of this problem to $GC[k]$.

First we reduce $GC[3]$ to $GC[4]$. Given a graph $G = (V, E)$, $V = \{1,\cdots,n\}$ with maximum degree 4, construct a graph $G' = (V', E')$ as follows. For each $i \in V$ let $\hat{i}$ be a new vertex and let $\{i, \hat{i}\}$ be an edge. Take $n-1$ copies of the complete graph K_3 on three vertices and connect all vertices of the jth copy to $\hat{j}$ and $\widehat{j+1}$, $1 \leq j \leq n-1$.

Let V' and E' be the resulting sets of vertices and edges and $G' = (V', E')$. Then the maximum degree in G' is 7. If G is 3-colorable, then G' is 4-colorable, by using a new color for the vertices $\hat{\imath}$ and the old colors for the K_3's.

Note that every 4-coloring of G' has to assign the same color to all vertices $\hat{\imath}$ because of the interconnecting K_3. Thus three colors suffice to color the vertices $i \in V$, whence G is 3-colorable.

The general reduction of $GC[k]$ to $GC[k+1]$ is similar. The interconnecting graphs are then K_k's. □

We proceed by proving a lemma that will be useful for the next theorem. Let $G = (V, E)$ be a graph with vertex degree at most $2k - 1$, $V = \{1, \cdots, n\}$. For $i \in V$ let Γ_i denote the set of neighbours of i including i. We say that a monomial m *selects* a variable x_i if $x_i \notin m$ and $x_j \in m$ for all $j \in \Gamma_i - \{i\}$, i.e., m contains all neighbours of i but not i itself.

We define our sample by the following positive and negative examples.

— $p_i := (1, \cdots, 1, 0, 1, \cdots, 1)$, $1 \leq i \leq n$ where the only 0 is at the ith position.

— For all $1 \leq i \leq n$ and all $T \subseteq \Gamma_i$ such that $|T| \geq 2$ and $i \in T$ the vector $\chi(\bar{T})$ is a negative example. $\chi(\bar{T})$ is the 0-1-vector having 0s at exactly those positions j with $j \in T$.

We let D^+ and D^- be uniform on these positive and negative examples and zero elsewhere.

LEMMA 7. *Let the sample be as above and let* $r = m_1 \oplus \cdots \oplus m_k$ *be a* k*-term-*RSE *with the property that it is not satisfied by any negative example, and, for some* $1 \leq l \leq n$, *there are positive examples* $p_{i_1}, \cdots, p_{i_l}$ *that satisfy* r. *Then, for each* $1 \leq \lambda \leq l$ *there exists a* $1 \leq j \leq k$ *such that* m_j *selects* x_{i_λ}.

Proof. Without loss of generality we may assume that $\{i_1, \cdots, i_l\} = \{1, \cdots, l\}$. Now for each $1 \leq i \leq l$ there is a monomial m not containing x_i, because $r(p_i) = 1$. Suppose, for a contradiction, that for some $i \in \{1, \cdots, l\}$, no m_j selects x_i, i.e., for all $1 \leq j \leq k$ if $x_i \notin m_j$, then there is some $t \in \Gamma_i - \{i\}$ such that $x_t \notin m_j$. Let $s(i)$ be the number of monomials m such that $x_i \notin m$. Note that $s(i)$ is odd because $r(p_i) = 1$ and $m(p_i) = 1$ if and only if $x_i \notin m$. Let $\mathcal{T}_i := \{T \subseteq \Gamma_i \mid i \in T, |T| \geq 2\}$. For $T \in \mathcal{T}_i$ let $q(i, T)$ be the number of monomials m such that: for all $t \in \Gamma_i : t \in T \iff x_t \notin m$, (i.e., such that exactly those neighbours of i which are in T are not in m). Then

$$s(i) = \sum_{T \in \mathcal{T}_i} q(i, T) .$$

As $s(i)$ is odd some $q(i, T)$ must be odd. Let T^* be maximal (with respect to $\subseteq$) such that $q(i, T^*)$ is odd. Recall that $u = \chi(\bar{T}^*)$ is the 0-1-vector which has 0s at exactly those positions t with $t \in T^*$. Therefore, $m(u) = 1$ for exactly those monomials m such that: $t \in T^* \Rightarrow x_t \notin m$. Let $s(i, T^*)$ denote their number. Then

$$s(i, T^*) = \sum_{\substack{T \supseteq T^* \\ T \in \mathcal{T}_i}} q(i, T) = q(i, T^*) + \sum_{\substack{T \supset T^* \\ T \in \mathcal{T}_i}} q(i, T) .$$

$q(i, T^*)$ is odd and, by maximality of T^*, all other $q(i, T)$ appearing in the sum are even. Hence $s(i, T^*)$ is odd. But this implies that an odd number of monomials is satisfied by u. Hence $r(u) = 1$, contradicting the consistency of r. □

Remark. Lemma 7 implies the existence of a subgraph of G of size at least l which is k-colorable. Indeed, for each node i, $1 \leq i \leq l$, choose the minimal j such that m_j selects x_i and color i with j. Then for all its neighbours $t \in \Gamma_i - \{i\}$ we have $x_t \in m_j$,

whence they receive a color different from j. Thus the monomials of the consistent k-*term*-RSE r induce a k-coloring on a subgraph of size at least l.

The next theorem has been independently proven by Blum and Singh [BS 90].

THEOREM 8. *For $k \geq 3$, k-term-RSE is not learnable, if $RP \neq NP$.*

Proof. Let $k \geq 3$. We show that the consistency problem for k-*term*-RSE is NP-hard using a reduction from $GC[k]$.

Given some instance G of $GC[k]$ let the sample and the distributions be defined as before Lemma 7. We first show that a k-coloring of G is polynomial-time convertible into a k-*term*-RSE that is consistent with the sample. Let U_j be the set of vertices colored by j, $1 \leq j \leq k$. Define monomials by

$$m_j := x_{\{1,\cdots,n\}-U_j}, \qquad 1 \leq j \leq k.$$

Let $r := m_1 \oplus \cdots \oplus m_k$. Then r is consistent with the sample: For all $1 \leq j \leq k$, $1 \leq l \leq n$ we have $m_j(p_l) = 1$ if and only if $l \in U_j$. Hence exactly one of the monomials is satisfied by p_l, whence $r(p_l) = 1$.

Let $i \in V$ and $T \subseteq \Gamma_i$, $|T| \geq 2$, $i \in T$. Then $u := \chi(\bar{T})$ is a negative example. Every vertex $i' \neq i$ in T is colored differently from i. Consider m_j for some $1 \leq j \leq k$ and recall that $x_l \in m_j$ if and only if $l \notin U_j$. Hence either $x_i \in m_j$ or $x_{i'} \in m_j$. The example u assigns 0 to both x_i and $x_{i'}$. Hence $m_j(u) = 0$, $1 \leq j \leq k$, and thus $r(u) = 0$. This shows that r is consistent with the sample.

Our next aim is to show that any k-*term*-RSE that is consistent with the sample induces a k-coloring of G. Let $r = m_1 \oplus \cdots \oplus m_k$ where each m_j is a monomial over $x_1, \cdots, x_n$. Assume that r is consistent with the sample. Then, by Lemma 7, applied with $l = n$, G is k-colorable.

There is only a polynomial number of examples, and the above reduction is certainly performed in polynomial time. □

4. Learnability from negative examples only. We have seen in §2 that k-RSE is learnable. Our next aim is to show that any learning algorithm for k-RSE needs both positive and negative examples. We start by proving two lemmas that are needed in this section.

LEMMA 9. *Let $\mathcal{C}$ and $\mathcal{C}'$ be classes of representations.*

(a) *If $f \in \mathcal{C} \Rightarrow \neg f \in \mathcal{C}'$, then any pos-prediction algorithm A^+ for $\mathcal{C}'$ can be converted into a neg-prediction algorithm A^- for $\mathcal{C}$.*

(b) *If, additionally, $f \in \mathcal{C}' \Rightarrow \neg f \in \mathcal{C}$, then any pos-learning algorithm A^+ for $\mathcal{C}'$ can be converted into a neg-learning algorithm A^- for $\mathcal{C}$.*

Proof. (a) Let $f \in \mathcal{C}$ (then $\neg f \in \mathcal{C}'$). Now construct some hypothesis g for $\neg f$ via A^+ treating negative examples for f as positive examples for $\neg f$. Then output $\neg g$.

(b) Proceed as in part (a) of the proof and note that $g \in \mathcal{C}'$ and $\neg g \in \mathcal{C}$. □

Remark. Note that the negation of a k-RSE is again a k-RSE because $\neg f = 1 \oplus f$. Negating a k-*term*-RSE in the same way leads to a $(k+1)$-*term*-RSE.

LEMMA 10. *Let $k \geq 1$ and let f be a k-RSE on n variables. Then one of the following cases holds:*

(a) *$f(v) = 0$, for all v,*

(b) *$f(v) = 1$, for all v,*

(c) *$|f^{-1}(0)| \geq 2^n/2^k$ and $|f^{-1}(1)| \geq 2^n/2^k$.*

For the next proofs we need the following definition.

DEFINITION. Let $m = \wedge_{i \in I} x_i^{\alpha_i}$, $I \subseteq \{1, \cdots, n\}$, $\alpha_i \in \{0, 1\}$, be a monomial. Then

$$C_m := \{v \in \{0,1\}^n \mid m(v) = 1\}$$

is the subcube of $\{0,1\}^n$ that is determined by fixing to 0 (1) those variables that appear (un)negated in m.

Proof (*of Lemma* 10). The proof is by induction on n.

$n = k$. If f is not constant on $\{0,1\}^n$, then $|f^{-1}(0)|, |f^{-1}(1)| \geq 1 = 2^n/2^k$.

$n > k$. Assume that f is not constant on $\{0,1\}^n$. We shall assume that the term **1** does not appear in f and the proof of the other case is similar.

Case 1. There is some variable, say x_n, which is contained in all monomials of f. Then $f_{|x_n=0} \equiv 0$, whence $|f^{-1}(0)| \geq 2^{n-1} \geq 2^n/2^k$. If $f_{|x_n=1} \equiv 1$ then $|f^{-1}(1)| = 2^{n-1}$ and we are done. Thus assume that $f_{|x_n=1}$ is a nonconstant $(k-1)$-RSE on the $(n-1)$-dimensional subcube C_{x_n}. Then, by induction, $|f^{-1}_{|x_n=1}(1)| \geq 2^{n-1}/2^{k-1} = 2^n/2^k$, whence $|f^{-1}(1)| \geq 2^n/2^k$.

Case 2. There is some variable, say x_n, which is contained in all monomials of length k, but there is some monomial of length less than k that does not contain x_n. Then $f_{|x_n=0}$ is a nonconstant $(k-1)$-RSE on $C_{\bar{x}_n}$ that, by induction, has at least $2^{n-1}/2^{k-1} = 2^n/2^k$ elements in each of $f^{-1}_{|x_n=0}(0)$ and $f^{-1}_{|x_n=0}(1)$.

Case 3. There is no variable that is contained in all monomials of length k. Note that this implies that $f_{|x_n=0}$ is a nonconstant k-RSE on the $(n-1)$-dimensional subcube $C_{\bar{x}_n}$. Hence, by induction, $|f^{-1}_{|x_n=0}(1)| \geq 2^{n-1}/2^k$ and $|f^{-1}_{|x_n=0}(0)| \geq 2^{n-1}/2^k$. The same considerations apply to $f_{|x_n=1}$ and C_{x_n}. Hence $|f^{-1}(0)|, |f^{-1}(1)| \geq 2(2^{n-1}/2^k)$. □

THEOREM 11. *For all $k \geq 1$, k-RSE is not neg-learnable and not pos-learnable even if the error bounds are constant.*

Proof. We only prove that k-RSE is not neg-learnable; the other claim follows from Lemma 9.

Let $m := \bar{x}_1\bar{x}_2\cdots\bar{x}_{k+1}$. Then C_m is the subcube of $\{0,1\}^n$, which consists of those vectors that have 0s in the first $k+1$ entries. We consider the following k-RSEs: $c_i := x_i$, $1 \leq i \leq k+1$. Note that each $c_i \equiv 0$ on C_m. Let D^- be uniform on C_m. For c_i we define D_i^+ to be uniform on $c_i^{-1}(1) = C_{x_i}$. We shall show that every k-RSE learned from negative examples will have a large error relative to one of the c_i. The idea is that negative examples drawn according to D^- cannot discriminate between the c_i and thus any hypothesis h produced from negative examples has to have a large error relative to one of them.

Let $\varepsilon = 2^{-(k+1)}$, $\delta = \frac{1}{k+2}$, and assume that there is a neg-learning algorithm for k-RSE. Let h be the hypothesis output by the algorithm.

Case 1. $h \not\equiv 0$ on C_m. Then, by Lemma 10, $h(v) = 1$ for a fraction of at least 2^{-k} of all $v \in C_m$. Hence, by uniformity of D^-, we have that the error relative to each c_i is at least $2^{-k} > \varepsilon$.

Case 2. $h \equiv 0$ on C_m. By Lemma 10, we know that $|h^{-1}(0)| \geq 2^{n-k}$. As $|C_m| = 2^{n-(k+1)}$ there are at least $2^{n-(k+1)}$ many vectors of $h^{-1}(0)$ outside C_m. Then there is some $i \in \{1, \cdots, k+1\}$ such that C_{x_i} contains a vector from $h^{-1}(0)$. Hence, by Lemma 10, h will compute 0 on a fraction of at least 2^{-k} of all vectors of C_{x_i}. Thus there is a concept c_i such that the probability that the algorithm outputs a hypothesis h whose error relative to c_i is at least $2^{-k} > \varepsilon$, is at least $\frac{1}{k+1} > \delta$. □

COROLLARY 12. 1-RSE *is neither neg-learnable nor pos-learnable, even if one allows arbitrary running time of the algorithm and a k-RSE to be produced as a hypothesis, $k \geq 1$.*

Proof. The concepts c_i to be learned in the proof of Theorem 11 are 1-RSE. The proof does not make any assumption about the size of the sample or the running time. □

5. Predictability. Our next aim is to prove predictability of k-RSE from one type of example. We first prove a lemma on the algebraic structure of the set of negative examples of a 1-RSE*. Let $v, b_1, \cdots, b_s \in \{0,1\}^n$. Then v is a *linear combination* (over $GF(2)$) of the b_i's if there are $\alpha_i \in \{0,1\}$, $1 \leq i \leq s$, such that $v = \sum_{i=1}^{s} \alpha_i b_i$. Let $\langle b_1, \cdots, b_s \rangle$ denote the set of linear combinations of the b_i's (i.e., the subspace spanned by the b_i's).

LEMMA 13. *For every $f \in$ 1-RSE*, the set of negative examples $f^{-1}(0)$ is a linear subspace of $\{0,1\}^n$.*

Proof. The negative examples are exactly those vectors that have an even number of ones at those positions j with $f_j = 1$, $1 \leq j \leq n$. This property also holds for any sum of two or more such vectors. □

THEOREM 14. 1-RSE* *is pos-predictable and neg-predictable.*

Proof. We prove the neg-predictability; the pos-predictability follows from Lemma 9. Let $f = \bigoplus_{i=1}^{n} f_i x_i$, $f_i \in \{0,1\}$, $0 \leq i \leq n$, be a 1-RSE*, and let D^+, D^- be distributions on the positive, respectively, negative examples of f. Let $q_1, \cdots, q_m \in f^{-1}(0)$ be a sample of negative examples. We have to produce in time $p(m,n)$ (for some polynomial p) a hypothesis h that is consistent with the sample, is of size polynomial in the number of variables (i.e., $\|h\| \leq q(n)$, q some polynomial, and $\|h\|$ does not depend on the size of the sample), and has a *single-sided error*, i.e.,

$$\sum_{h(v)=0} D^+(v) = 0.$$

This is a sufficient condition for predictability (as is implicitly proved in [BEHW 87]).

We shall formulate an algebraic hypothesis, namely, h will be a basis of a subspace of $\{0,1\}^n$.

To this end choose a maximal linearly independent set in the sample, say, $b_1, \cdots, b_s$, $s \leq n$. Then our hypotheses will be a description of $H := \langle b_1, \cdots, b_s \rangle$, e.g., $\{b_1, \cdots, b_s\}$ is such a description, whence $\|h\| = O(n^2)$.

The error is single-sided because $H \subseteq f^{-1}(0)$. □

COROLLARY 15. k-RSE *is pos-predictable and neg-predictable, $k \geq 1$.*

Proof. The substitution theorem of [KLPV 87a] does not only hold for learning algorithms but for prediction algorithms as well. Then the corollary follows from Theorem 14. The substitution also yields the algorithm for the resulting class. □

Theorem 14 can be strengthened to learnability if the distributions D^+ and D^- are uniform.

THEOREM 16. *Let $f \in$ 1-RSE, let D^+ and D^- be uniform on $f^{-1}(1)$ and $f^{-1}(0)$, respectively. Then* 1-RSE *is neg-learnable and pos-learnable. In addition, the learning algorithm identifies the original concept f with probability at least $1 - \delta$.*

Proof. We first investigate the case of a 1-RSE*, i.e., without the term 1. By Lemma 9 it suffices to show neg-learnability. Let $\delta > 0$. We draw m negative examples $q_1, \cdots, q_m$ and construct a maximal independent set $\{b_1, \cdots, b_s\}$, by collecting q_j's that are linearly independent. The q_j are drawn according to the uniform distribution. Hence the probability for some q_j to be linearly independent of $\{b_1, \cdots, b_s\}$ is at least $\frac{1}{2}$ as long as the full dimension of $f^{-1}(0)$ has not yet been reached. The expected number of vectors that we have to draw until the full dimension is reached is thus at most $2n$. Then $\lceil 1/\delta \rceil \cdot 2n$ negative examples suffice to make the error less than δ. In order to see this, let X be the random variable "number of examples until the full

dimension is reached" and assume that $Pr[X \geq (1/\delta)2n] > \delta$. Then

$$\begin{aligned} E[X] &= \sum_{m=1}^{\infty} m \cdot Pr[X = m] \geq \sum_{m=(1/\delta)2n}^{\infty} m \cdot Pr[X = m] \\ &\geq \sum_{m=(1/\delta)2n}^{\infty} \frac{1}{\delta}2n \cdot Pr[X = m] = \frac{1}{\delta}2n \sum_{m=(1/\delta)2n}^{\infty} Pr[X = m] \\ &= \frac{1}{\delta}2n \cdot Pr\left[X \geq \frac{1}{\delta}2n\right] > \frac{1}{\delta}2n \cdot \delta = 2n. \end{aligned}$$

Hence we get the contradiction that the expected number is strictly larger than $2n$.

The negative examples form a linear subspace. We want to check whether the variable x_i is a term in the 1-RSE* f. To this end we test for each canonical basevector e_i, $1 \leq i \leq n$, whether $e_i \in \langle b_1, \cdots, b_s \rangle$. Then x_i is a term in f if and only if this is not the case.

Let us now assume that we have a 1-RSE that contains the term 1. Then the negative examples contain an odd number of 1s at the relevant positions. This property is preserved under linear combinations of an odd number of such examples. Any example containing an even number of 1s at the relevant positions is positive. We use a technique similar to the one described above to keep track of the "space" of odd linear combinations.

In order to find out whether the target concept contains the term 1 we draw some more examples and check whether one of them is a linear combination of an even number of the old examples. If so, we know that the term 1 is present, otherwise we assume it is not. On the average, two additional examples will suffice. Hence the sample size needed becomes $\lceil 1/\delta \rceil \cdot (2n + 2)$. □

This result cannot be carried over to k-RSE using the substitution theorem of [KLPV 87a] because substitution does not preserve uniform distribution.

We conclude with some results on predicting k-*term*-RSE.

COROLLARY 17. 1-*term*-RSE *is not neg-predictable, but pos-learnable.* k-*term*-RSE *is neither pos-predictable nor neg-predictable for* $k \geq 2$.

Proof. In [KLPV 87a] it is proven that the class of monotone monomials is not neg-predictable but pos-learnable. This class is exactly the class 1-*term*-RSE. k-*term*-RSE, $k \geq 2$ is not neg-predictable because 1-*term*-RSE is not. By the remark following Lemma 9, if k-*term*-RSE is pos-predictable, then $(k-1)$-*term*-RSE is neg-predictable. Hence k-*term*-RSE cannot be pos-predictable, $k \geq 2$. □

If both types of examples are allowed, even 2-*term*-RSE is predictable.

COROLLARY 18. 2-*term*-RSE *is predictable.*

Proof. Consider the 2-*term*-RSE $r = m_1 \oplus m_2$, where m_1, m_2 are monotone monomials. Then

$$m_1 \oplus m_2 = (m_1 \vee m_2) \wedge \neg(m_1 \wedge m_2) = (m_1 \vee m_2) \wedge c, \quad \text{where } c = \neg(m_1 \wedge m_2)\ .$$

Now the term $(m_1 \vee m_2)$ can be transformed into a 2-CNF using the distributive law while c becomes a 1-DNF using deMorgans's law. The class k-CNF is pos-learnable with single-sided error and k-DNF is learnable (see [V 84]). The conjunction of a concept class that is pos-learnable with single-sided error and one that is learnable is itself learnable (see [KLPV 87a]). □

6. Sample size and worst case mistake bounds. In this paper we presented two algorithms concerning k-RSE. Both of them have as input a sample and the error

parameters. One of them (Corollary 3) produces a consistent k-RSE, the other one (Corollary 15), a consistent hypothesis that is not a k-RSE. Both algorithms can be converted into learning or prediction algorithms that are optimal in two respects: sample size and mistake bound.

In order to prove these results we have to know the Vapnik–Chervonenkis (VCdim) dimension of the class k-RSE. The Vapnik–Chervonenkis dimension of a concept class $\mathcal{C}$ (of Boolean formulas) is defined as follows: We say that a set $S \subseteq \{0,1\}^n$ is *shattered by* $\mathcal{C}$ if for every $U \subseteq S$ there is a formula $f \in \mathcal{C}$ such that $f(u) = 1$ for $u \in U$, and $f(u) = 0$ for $u \in S - U$. The *Vapnik–Chervonenkis dimension*, VCdim($\mathcal{C}$), of $\mathcal{C}$ is the cardinality of a largest set that is shattered by $\mathcal{C}$. Note that, for finite $\mathcal{C}$, $\text{VCdim}(\mathcal{C}) \leq \log|\mathcal{C}|$.

We claim that VCdim(1-RSE) is $n+1$, and that VCdim(k-RSE) is $\Theta(n^k)$. Recall that $|\text{1-RSE}| = 2^{n+1}$, whence $n+1$ is an upper bound on VCdim(1-RSE). On the other hand, the set $S = \{z, e_1, \cdots, e_n\}$ is shattered by 1-RSE, where z is the zero vector and e_i, $i \geq 1$, is the ith canonical base vector. Let U be a subset of S not containing z, say, $U = \{e_{i_1}, \cdots, e_{i_k}\}$, $k \leq n$. Then $x_{i_1} \oplus \cdots \oplus x_{i_k}$ computes 1 on U and 0 on $S - U$. If U contains z then consider $S - U = \{e_{j_1}, \cdots, e_{j_{n-k}}\}$ and note that $1 \oplus x_{j_1} \oplus \cdots \oplus x_{j_{n-k}}$ computes 1 on U and 0 on $S - U$. Hence VCdim(1-RSE) $= n+1$. A similar argument shows that VCdim(k-RSE) $= \Omega(n^k)$. The following set of size $\binom{n}{k}$ is shattered by k-RSE: $S = \{v \in \{0,1\}^n | v \text{ contains exactly } k \text{ ones }\}$. Given $v = (v_1, \cdots, v_n)$, let m_v be the monomial containing exactly those variables x_i with $v_i = 1$. Then, for every $U \subseteq S$, the k-RSE defined by $\bigoplus_{v \in U} m_v$ is identically 1 on U and identically 0 on $S - U$. On the other hand, $|\mathcal{C}| = 2^{O(n^k)}$. Thus VCdim($k$-RSE) $= \Theta(n^k)$.

Now the conversion of the algorithm of Corollary 3 (respectively, Corollary 15) to a learning (respectively, prediction) algorithm A is done as follows.

(1) Construct a sample X of size $m_A(n, \varepsilon, \delta)$ that is sufficiently large.

(2) Output a hypothesis consistent with X.

It then follows from results of Vapnik and Blumer and others (see, e.g., [EHKV 89]) that

$$m_A(n, \varepsilon, \delta) = O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta} + \frac{n^k}{\varepsilon}\right)$$

is an appropriate choice. Moreover, this is optimal up to a constant factor. For details, see [EHKV 89].

Another nice feature of our algorithms for k-RSE is that they are convertible into *on-line prediction algorithms*, i.e., algorithms that are applied to an infinite sequence of examples. For any new example they first have to output a prediction whether it is positive or negative. Afterwards the correct answer is revealed. The *mistake bound* of such an algorithm is the maximum number of false predictions made on any sequence of examples. The *best worst case mistake bound* of a concept class $\mathcal{C}$, denoted by $\text{OPT}_{\mathcal{C}}(n)$, is the minimum of the mistake bounds taken over all possible on-line prediction algorithms for $\mathcal{C}$ (regardless of their running times). For more details, see [L 87].

For instance, our learning algorithm A for 1-RSE* can be converted to an on-line prediction algorithm in the following way: The algorithm keeps track of the subspace S spanned by the negative examples $x = (x_1, \cdots, x_n)$ seen so far. Initially S is set to $\{0\}$. In order to make a prediction on a new example x the algorithm checks whether x is in S. If so, A can be used to produce a consistent hypothesis h. The prediction is $h(x)$, which is always correct. Otherwise, the prediction is 1 (default value), which is possibly wrong. Whenever a wrong prediction is made the dimension of S is increased by 1. Therefore the total number of mistakes is at most $n + 1$.

Again the substitution method of [KLPV 87a] can be applied to show that the total number of mistakes of the corresponding on-line prediction algorithm for k-RSE is $O(n^k)$. Moreover, we can achieve that predictions on positive examples are always correct, using a modification of the neg-prediction algorithm of Corollary 15.

These bounds can be shown to be tight using results of Littlestone [L 87]. There, it is proved that the Vapnik–Chervonenkis dimension of a concept class $\mathcal{C}$ is a lower bound on $\mathrm{OPT}_{\mathcal{C}}$, while, for finite $\mathcal{C}$, $\log|\mathcal{C}|$ is an upper bound. We summarize the above results in the following theorem.

THEOREM 19. *For k-RSE the best worst case mistake bounds are* $\mathrm{OPT}_{1\text{-RSE}}(n) = n+1$ *and* $\mathrm{OPT}_{k\text{-RSE}}(n) = \Theta(n^k)$, *where n is the number of variables.*

7. Summary and conclusions. Table 1 shows the results on learning and predicting k-RSE and k-*term*-RSE.

TABLE 1

	Learnable	pos-learnable	neg-learnable	Predictable	pos-predictable	neg-predictable
k-RSE, $k \geq 1$	yes	no	no	yes	yes	yes
1-*term*-RSE	yes	yes	no	yes	yes	no
2-*term*-RSE	no[1]	no	no	yes	no	no
k-*term*-RSE, $k \geq 3$	no[1]	no	no	yes[2]	no	no

[1] The result is proved using the assumption $RP \neq NP$.
[2] See Blum and Singh [BS 90] .

We would like to mention that some of the above results can be extended to c-heuristic learnability. We say that $\mathcal{C}$ *is c-heuristically learnable by $\mathcal{H}$*, $c \in (0,1)$, c *rational*, if there exists a polynomial p and a learning algorithm A such that for all n, all $f \in \mathcal{C}_n$, all distributions D^+, D^-, and all $\varepsilon > 0$ and $\delta > 0$,
— L halts in time $p(n, \|f\|, \frac{1}{\varepsilon}, \frac{1}{\delta})$,
— L outputs a formula $g \in \mathcal{H}_n$ that, with probability at least $(1-\delta)$, has the property:

$$\sum_{v \in g^{-1}(0)} D^+(v) < 1 - c \quad \text{and} \quad \sum_{v \in g^{-1}(1)} D^-(v) < \varepsilon\ .$$

If no such $g \in \mathcal{H}$ exists then A outputs some default concept.

If g is not the default concept then g correctly classifies a fraction of at least c of all positive examples of f and the error made on the negative ones is very small.

The proof of the next theorem can be found in [FS 90].

THEOREM 20. *For all $c \in (0,1)$, c rational and all $k \geq 1$, the class* RSE *of all* RSE-*formulae is not c-heuristically learnable by k-term*-RSE.

REFERENCES

[BEHW 87] A. BLUMER, A. EHRENFEUCHT, D. HAUSSLER, AND M. WARMUTH, *Occam's razor*, Inform. Process. Lett., 24 (1987), pp. 377–380.

[BS 90] A. BLUM AND M. SINGH, *Learning functions of k terms*, in Proc. Third Annual Workshop on Computational Learning Theory (COLT 90), Morgan Kaufmann, San Mateo, CA, 1990, pp. 144–153.

[EHKV 89] A. EHRENFEUCHT, D. HAUSSLER, M. KEARNS, AND L. VALIANT, *A general lower bound on the number of examples needed for learning*, Inform. Comput., 82 (1989), pp. 247–261.

[FS 90] P. FISCHER AND H. U. SIMON, *On learning ring-sum-expansions*, in Proc. Third Annual Workshop on Computational Learning Theory (COLT 90), Morgan Kaufmann, San Mateo, CA, 1990, pp. 130–143.

[GJ 79] M. GAREY AND D. JOHNSON, *Computers and Intractability*, W. H. Freeman, New York, 1979.

[HSW 90] D. HELMBOLD, R. SLOAN AND M. WARMUTH, *Learning integer lattices*, in Proc. Third Annual Workshop on Computational Learning Theory (COLT 90), Morgan Kaufmann, San Mateo, CA, 1990, pp. 288–302.

[KLPV 87] M. KEARNS, M. LI, L. PITT, AND L. VALIANT, *Recent results on Boolean concept learning*, Workshop on Machine Learning, University of California, Irvine, CA, 1987, pp. 337–352.

[KLPV 87a] ———, *On the learnability of Boolean formulae*, in Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 285–295.

[PV 88] L. PITT AND L. VALIANT, *Computational limitations on learning from examples*, J. Assoc. Comput. Mach., 35, (1988), pp. 965–984.

[L 87] N. LITTLESTONE, *Learning quickly when irrelevant attributes abound: A new linear threshold algorithm*, in Proc. 28th IEEE Symposium on Foundations of Computer Science, 1987, pp. 68–77.

[V 84] L. VALIANT, *A theory of the learnable*, Comm. ACM, 27 (1984), pp. 1134–1142.

[W 87] I. WEGENER *The Complexity of Boolean Functions*, Wiley–Teubner, Stuttgart, Germany, 1987.

SIAM J. COMPUT.
Vol. 21, No. 1, pp. 193–201, February 1992

© 1992 Society for Industrial and Applied Mathematics
015

LOCALITY IN DISTRIBUTED GRAPH ALGORITHMS*

NATHAN LINIAL†

Abstract. This paper concerns a number of algorithmic problems on graphs and how they may be solved in a distributed fashion. The computational model is such that each node of the graph is occupied by a processor which has its own ID. Processors are restricted to collecting data from others which are at a distance at most t away from them in t time units, but are otherwise computationally unbounded. This model focuses on the issue of *locality* in distributed processing, namely, to what extent a global solution to a computational problem can be obtained from locally available data.

Three results are proved within this model:

- A 3-coloring of an n-cycle requires time $\Omega(\log^* n)$. This bound is tight, by previous work of Cole and Vishkin.
- Any algorithm for coloring the d-regular tree of radius r which runs for time at most $2r/3$ requires at least $\Omega(\sqrt{d})$ colors.
- In an n-vertex graph of largest degree Δ, an $O(\Delta^2)$-coloring may be found in time $O(\log^* n)$.

Key words. distributed algorithms, graph theory, locality, lower bounds

AMS(MOS) subject classifications. 05C35, 68R10, 68Q99

1. Introduction. In distributed processing all computations are made based on local data. The aim of this paper is to bring up limitations that follow from this local nature of the computation. Note that within the various computational models for parallel computers this difficulty is specific to the distributed model. Shared memory allows for fast dissemination of data, but no such means exist when dealing with distributed systems.

In the present paper we are mostly interested in proving lower bounds, and therefore assume a powerful version of the distributed model: Each node of the undirected graph $G = (V, E)$ is occupied by a processor. Computation is completely synchronous and reliable. At each time unit a processor may pass messages to each of its neighbors, and message size is unrestricted. Also, any computations carried out by individual processors take one time unit and are not restricted in any way. This paper is only concerned with the radius of the neighborhood around each node from which data may be collected, this radius being the only significant parameter in this model, as we later elaborate. Of interest is the time complexity of various "global" functions of G, and the concrete examples are coloring and finding maximal independent sets. Thus the theme of this paper is how local data may be utilized to find globally defined solutions.

Before we proceed, symmetry-breaking has to be addressed (see [JS] and the references therein for literature on symmetry-breaking). It is well known that most functions cannot be computed in a distributed fashion by anonymous processors, even for very simple graphs G. This impossibility usually results from symmetries that G may have. Such symmetries are usually broken by means of either randomization or the use of IDs, and the present paper is concerned only with the latter. Thus we assume that there is a mapping ID from the set of vertices V to the positive integers.

---

* Received by the editors July 17, 1989; accepted for publication (in revised form) December 4, 1990.

† Computer Science Department, Hebrew University, Jerusalem 91904, Israel. Part of this work was done while the author was visiting the IBM Research Center, Almaden, California, and the Department of Computer Science, Stanford University, Stanford, California 94305. This work was supported in part by grants from the Israel–US Binational Science Foundation and the Israeli Academy of Sciences.

In most cases ID is a bijection onto $1, \cdots, |V|$. It is assumed that at time zero the processor occupying a node in G knows the ID of that node. Incidentally, all our lower bounds hold even if every processor knows in advance what the graph G is, and only the labeling function ID is unknown.

It is clear that the present model allows us to compute every function of G in time $O(\text{diameter}(G))$. After this amount of time every processor obtains complete knowledge of both G and ID. The problem is thus solved if, in the memory of each processor a solution for the entire problem is prestored, for every possible labeling. Our concern is therefore only with time complexities below $\text{diam}(G)$. The major question we raise is which functions may be computed by a nontrivial algorithm in this model, in the sense that they can be computed faster than $\text{diam}(G)$.

The model proposed here is, of course, of purely theoretical interest. Of the many difficulties arising in distributed processing, it focuses only on transforming local data into a global solution. Further research into this model may help classify problems as either locally computable (solvable in time shorter than $\text{diam}(G)$) or not. One may also look for bounds on run times which depend on graph parameters other than the diameter. In accordance with the theory of the complexity class NC, it seems natural to investigate graph problems whose time complexity in this model is polylogarithmic in the number of vertices.

Here are our main results:

(1) Finding a maximal independent set distributively in a labeled n-cycle, requires time $\Omega(\log^* n)$. This bound is tight in view of the $O(\log^* n)$ algorithm by Cole and Vishkin [CV]. (Technically, their result was stated in the PRAM model, but it extends without change to the distributed model as well.) Our proof relies on the interesting construct of *neighborhood graphs*. An alternative proof based on the Ramsey theorem was found by a number of other investigators in the area [A].

(2) Coloring trees: Let T be the d-ary tree of height r. In time $2r/3$, it is impossible to color T in fewer than $\sqrt{d}$ colors. Note in contrast the algorithm by Goldberg and Plotkin [GP], which shows that if every node in T "knows its parent in the tree," i.e., a consistent orientation from the root outwards is given, then a 3-coloring can be found in time $O(\log^* n)$. Though stated for the PRAM model, it is easy to see that this result of [GP] applies to the distributed model as well.

(3) In a labeled graph of order n with maximal degree Δ, it is possible to find an $O(\Delta^2)$-coloring in time $(1 + o(1)) \log^* n$. This was previously shown in [GP] only in the case of constant Δ.

Our terminology is standard: a *k-labeling* of a graph $G = (V, E)$ is 1:1 mapping $f : V \to \{1, \cdots, k\}$. In case $k = |V|$ a k-labeling is called a *labeling*. Given a k-labeling, G is said to be k-labeled, etc. Logarithms are to base 2. The k times iterated logarithm is denoted by $\log^{(k)} x$, i.e., $\log^{(1)} x := \log x$ and $\log^{(k)} x = \log(\log^{k-1} x)$. The least integer k for which $\log^{(k)} x < 1$ is denoted by $\log^* x$.

2. Lower bound on finding a maximal independent set in a cycle. In [CV] a very nice algorithm was presented to find a maximal independent set of vertices (= MIS) in the n-cycle C_n in time $\log^* n$. In this section we show that this is optimal even in the present model, where computation takes no time. The algorithm presented in §4 also achieves this time bound.

A basic observation is that in the present model there is no loss of generality in assuming that processing proceeds by first collecting all data and then deciding. That is, at time t each processor knows the labeling of all nodes at distance t or less away. Also known are all edges between these nodes, except for edges both endpoints

of which are at distance exactly t. Note that no further information can reach a processor by time t. This allows us to view the problem in purely combinatorial terms.

Let us state our theorem.

THEOREM 2.1. *A synchronous distributed algorithm which finds a maximal independent set in a labeled n-cycle must take at least* $\frac{1}{2}(\log^* n - 1)$ *units of time. An algorithm of the same class which colors the n-cycle with three colors requires time at least* $\frac{1}{2}(\log^* n - 3)$. *The same bounds hold also for randomized algorithms.*

Proof. The proof holds even under the assumption that there is a consistent notion of clockwise orientation common to all processors. Given an algorithm which finds a maximal independent set in the n-cycle endowed with a clockwise orientation, it is easily seen that in one more timestep, the cycle may be 3-colored. The lower bound is established for 3-coloring.

Coming back to the previous observation, at time t the data known to a processor P is an ordered list of $2t+1$ labels, starting t places before it, through its own and on to the next t labels. Let V be the set of all vectors $(x_1, \cdots, x_{2t+1})$ where the x_i are mutually distinct integers from $\{1, \cdots, n\}$. The algorithm is nothing but a mapping c from V into $\{1, 2, 3\}$.

Let us denote by $B_{t,n}$ the graph whose set of vertices is V. All edges of $B_{t,n}$ are given by:

$$(x_1, \cdots, x_{2t+1}) \quad \text{and} \quad (y, x_1, \cdots, x_{2t})$$

are neighbors for all $y \neq x_{2t+1}$. So $B_{t,n}$ has $n(n-1)\cdots(n-2t)$ vertices and is regular of degree $2(n-2t-1)$. Note that the mapping $c : V \to \{1, 2, 3\}$ is, in fact, a proper 3-coloring of $B_{t,n}$. For suppose that c assigns

$$(x_1, \cdots, x_{2t+1}) \quad \text{and} \quad (y, x_1, \cdots, x_{2t})$$

the same color. Then the 3-coloring algorithm for the n-cycle fails in case the labeling happens to contain the segment:

$$y, x_1, x_2, \cdots, x_{2t+1}.$$

The proof follows now by standard graph-theoretic arguments which show that the chromatic number $\chi(B_{t,n})$ of $B_{t,n}$ satisfies

$$\chi(B_{t,n}) = \Omega(\log^{(2t)} n),$$

the $2t$ times iterated logarithm of n. Therefore, for $\chi(B_{t,n})$ to be at most 3, we must have $t = \Omega(\log^* n)$.

The lower bound on $\chi(B_{t,n})$ is proved, using a family of digraphs $D_{s,n}$ closely related to $B_{t,n}$. The vertices of $D_{s,n}$ are all sequences $(a_1, \cdots, a_s)$ with $1 \leq a_1 < a_2 < \cdots < a_s \leq n$. The outneighbors of $(a_1, \cdots, a_s)$ are all vertices of the form $(a_2, \cdots, a_s, b)$ with $a_s < b \leq n$. Note that $B_{t,n}$ contains the underlying graph of $D_{2t+1,n}$, so in particular $\chi(B_{t,n}) \geq \chi(D_{2t+1,n})$.

Given a digraph $H = (V, E)$ its *dilinegraph* $DL(H)$ is a digraph whose vertex set is E with (u, w) an edge if $head_H(u) = tail_H(w)$. The relation between the digraphs $D_{s,n}$ is given by Proposition 2.1.

PROPOSITION 2.1. $D_{1,n}$ *is obtained from the complete graph of order n by replacing each edge by a pair of edges, one in each direction, and* $D_{s+1,n} = DL(D_{s,n})$ *for all* $s \geq 1$.

Proof. The statement concerning $D_{1,n}$ is just the definition. For the other claim, identify the edge connecting $(x_1, \cdots, x_s)$ and $(x_2, \cdots, x_s, y)$ in $D_{s,n}$ with the vertex $(x_1, \cdots, x_s, y)$ in $V(D_{s+1,n})$ and check that the adjacency relationship in $D_{s+1,n}$ is that of $DL(D_{s,n})$. □

The bound on $\chi(D_{s,n})$ is derived from the following simple and well-known proposition.

PROPOSITION 2.2. *For a digraph G,*

$$\chi(DL(G)) \geq \log \chi(G).$$

Proof. A k-coloring of $DL(G)$ may be thought of as a mapping $\Psi : E(G) \to \{1, \cdots, k\}$ such that if $u, w \in E(G)$ and $head(u) = tail(w)$, then $\Psi(u) \neq \Psi(w)$. Now vertex-color G by associating with node x the set

$$c(x) = \{\Psi(u) \mid x = tail(u)\}.$$

This is easily seen to be a 2^k vertex-coloring of G. Indeed if $u = (x, y) \in E(G)$, then $\Psi(u) \in c(x)$ but $\Psi(u) \notin c(y)$, or else Ψ is improper. Therefore $c(x) \neq c(y)$. □

The main claim can be derived now. If an MIS can be found in time $t-1$, then necessarily

$$3 \geq \chi(B_{t,n}).$$

But

$$\chi(B_{t,n}) \geq \chi(D_{2t+1,n}) \geq \log^{(2t)} n.$$

So

$$2t \geq \log^* n - 1, \qquad t \geq \frac{1}{2}(\log 2^* n - 1),$$

as claimed.

The claim on randomized algorithms is proved in Corollary 2.1 below. □

In contrast with the low time complexity of 3-coloring, we can show that for an even n, finding a 2-coloring of C_n requires time $\Omega(n)$.

THEOREM 2.2. *A synchronous distributed algorithm which 2-colors a labeled $2n$ cycle with labels from $\{1, \cdots, 2n\}$ must take at least $n-1$ units of time.*

Proof. Now the lowest t has to be found such that $B_{t,2n}$ is bipartite. But even for $t = n-2$, the graph $B_{t,2n}$ contains an odd cycle:

$$(1, \cdots, 2t+1), (2, \cdots, 2t+2), (3, \cdots, 2t+3), (4, \cdots, 2t+3, 1),$$

$$(5, \cdots, 2t+3, 1, 2), \cdots, (2t+3, 1, \cdots, 2t), (1, \cdots, 2t+1).$$

The claim follows. □

Let us point out that for an even n the last theorem implies that finding a *maximum* independent set requires time $\lceil n/2 \rceil - 1$. It is easily verified that the same is true for odd n as well.

We want to elaborate on the method developed here and point out its general features. Given a graph $G = (V, E)$ of order n, and $t \geq 1$, the *t-neighborhood graph of*

$G, N_t(G)$ is constructed as follows: For every $x \in V$ let $S_t(x)$ be the subgraph of G spanned by those vertices y whose distance from x is at most t. For every x consider all the n-labelings of $S_t(x)$.

Every such labeling Ψ is a node in $N_t(G)$. Let $\Psi_1 : V_t(x) \to \{1, \cdots, n\}$ and $\Psi_2 : V_t(y) \to \{1, \cdots, n\}$ be two of these vertices of $N_t(G)$. They are taken to be neighbors in $N_t(G)$ if $[x, y] \in E(G)$ and there is a labeling $\Phi : V(G) \to \{1, \cdots, n\}$ such that

$$\Phi \mid S_t(x) = \Psi_1$$

and

$$\Phi \mid S_t(y) = \Psi_2.$$

Some easy observations regarding $N_t(G)$ follow.

PROPOSITION 2.3. *Neighborhood graphs have the following properties:*

(1) $\chi(N_t(G))$ *is the least number of colors with which* G *may be colored distributively on time* t.

(2) $\chi(N_t(G)) = \chi(G)$ *for* $t \geq \operatorname{diam}(G)$.

(3) $\chi(N_t(G))$ *is nonincreasing with* t.

(4) *For* $G = C_n$, *the graph* $B_{t,n}$ *is obtained from* $N_t(C_n)$ *by identifying vertices in* $N_t(C_n)$ *with identical sets of neighbors. In particular,*

$$\chi(B_{t,n}) = \chi(N_t(C_n)). \qquad \square$$

As in the case of the cycle, it is helpful to identify nonadjacent vertices with an identical set of neighbors. Such an operation is called a *reduction*. Clearly it does not change the chromatic number, while it may significantly simplify the graph. Neighborhood graphs seem to be very interesting and promising constructs. Unfortunately, the only case where we managed to calculate their graphical parameters is that of a cycle. Particularly interesting are cases where in G the graph $S_t(x)$ is independent of x, as is the case, for example, with vertex transitive graphs.

Proposition 2.3 yields the following easy but interesting corollary.

COROLLARY 2.1. *In the present model the time required to properly color a given graph with a given number of colors cannot be reduced by using randomization.*

Proof. The most general form of a randomized algorithm in the present model allows each processor to precede each round of computation with any number of coin flips, the outcomes of which are then passed to its neighbors along with all other information, as in the deterministic version. Consider the following different class $\mathcal{C}$ of randomized graph algorithms: A sufficiently long string of bits σ is first selected at random by some random source (not necessarily one of the processors in the graph) and then announced to all processors. From this point on, the algorithm proceeds in the usual deterministic way, where each processor gets to see its labeled t-neighborhood (as well as the random string σ, of course). It is not hard to see that any randomized algorithm which can be performed in the present model can be simulated by an algorithm in $\mathcal{C}$. This is because σ can be made long enough to include as many random bits as may be required in any run of the original algorithm by any processor at all. In the algorithm from $\mathcal{C}$, processors get the advantage of being informed of all these random bits, and in advance, which can only help.

Since we are interested in a lower bound, there is no loss of generality in considering only algorithms from the class $\mathcal{C}$. Such an algorithm running in time t entails a

function which computes a color for a vertex from its t-neighborhood and the random σ. Fix a graph G, an integer t, and two adjacent vertices x and y in G. Consider the adjacent vertices ξ and η in $N_t(G)$ which represent compatible labelings of t-neighborhoods of x, y, and a random string σ. The color is chosen based on seeing the labeled neighborhood ξ, and the random σ must differ from the one given for η, σ. In other words, the coloring function constitutes a valid coloring for a graph which is the disjoint union of copies of $N_t(G)$, one copy per each σ. This graph has, of course, the same chromatic number as $N_t(G)$, and the claim follows. □

3. Lower bound on coloring trees.

THEOREM 3.1. *Let $T = T_{d,r}$ be the rooted d-regular tree of radius r. Any synchronous distributed algorithm running in time $\leq \frac{2}{3}r$ cannot color T by fewer than $\frac{1}{2}\sqrt{d}$ colors. The same bound holds for randomized algorithms as well.*

Proof. There are d-regular graphs $R_{d,n}$ on n vertices of chromatic number $\geq \frac{1}{2}\sqrt{d}$, where all cycles have length $\geq (4/3)(\log n/\log(d-1))$ (see [LPS]). Consequently for $t < (2/3)(\log n/\log(d-1))$, the graph $N_t(T_{d,r})$ contains a copy of a reduction of $N_t(R_{d,n})$. Therefore

$$\chi(N_t(T_{d,r})) \geq \chi(N_t(R_{d,n})) \geq \chi(R_{d,n}) \geq \tfrac{1}{2}\sqrt{d},$$

and so $T_{d,r}$ cannot be colored with fewer than $\frac{1}{2}\sqrt{d}$ colors in time t. The conclusion follows now on observing that for $T_{d,r}$,

$$\frac{\log n}{\log(d-1)} \geq r.$$

Corollary 2.1 establishes the bound for randomized algorithms as well. □

Two remarks are in order now: It is probably possible to improve the lower bound of $\frac{1}{2}\sqrt{d}$ to $\Omega(d/\log d)$ by using an appropriate random graph rather than Ramanujan graphs (e.g., [B, §11.4]). It is shown in the next section that for a d-regular graph, an $O(d^2)$-coloring can be found in time $O(\log^* n)$. The gap between $(d/\log d)$ and d^2 is quite intriguing and is closely related to the complexity of finding an MIS distributively, as we explain below. Also, it is not clear how the number of colors sufficient to color the tree goes down as t grows from $(2r/3)$ to $2r$, where already two colors suffice.

4. An $O(\log^* n)$ algorithm for $O(\Delta^2)$-coloring. This section contains some positive results on what can be achieved distributively in this model. It is shown how to find an $O(\Delta^2)$-coloring in time $O(\log^* n)$. The first proof is based on the existence of certain hypergraphs for which no explicit construction is known. However, as Karloff pointed out [K], a slightly weaker, though constructive, set system suffices to achieve this goal.

THEOREM 4.1. *Let G be a graph of order n and largest degree Δ. It is possible to color G with $5\Delta^2 \log n$ colors in one unit of time distributively. Equivalently,*

$$\chi(N_1(G)) \leq 5\Delta^2 \log n.$$

Proof. We need some combinatorial preparation.

LEMMA 4.1. *For integers $n > \Delta$, there is a family J of n subsets of* $\{1, \cdots, 5\lceil \Delta^2 \log n \rceil\}$ *such that if* $F_0, \cdots, F_\Delta \in J$, *then*

$$F_0 \not\subseteq \bigcup_1^\Delta F_i.$$

Proof. The existence of such families was considered in the literature [KSS], [EFF], and follows, e.g., from Theorem 3.1 in [EFF]. This lemma is simple enough, though, for a proof to be reproduced here. To prove the lemma, let $m = 5\lceil \Delta^2 \log n \rceil$ and consider a random collection J of n subsets of $\{1, \cdots, m\}$ which is constructed as follows: For $1 \le x \le m$ and $1 \le i \le n$, let $Pr(x \in S_i) = (1/\Delta)$. All the decisions on whether $x \in S_i$ are made independently.

We claim that there is a selection of such a family for which the lemma holds. The probability that for a given $1 \le x \le m$ and given $F_0, \cdots, F_\Delta \in J$ (i.e., $F_i = S_{\mu_i}$ for appropriately chosen indices), there holds

$$x \in F_0 \setminus \left(\bigcup_1^\Delta F_i \right)$$

is

$$\frac{1}{\Delta}\left(1 - \frac{1}{\Delta}\right)^\Delta \ge \frac{1}{4\Delta}.$$

Therefore, the probability that $F_0 \subseteq \cup_1^\Delta F_i$ is at most $(1 - (1/4\Delta))^m$. The number of ways for choosing $F_0, \cdots, F_\Delta$ is $(\Delta + 1)\binom{n}{\Delta+1}$. Therefore if

$$\left(1 - \frac{1}{4\Delta}\right)^m (\Delta + 1)\binom{n}{\Delta + 1} < 1,$$

then there is a selection of a family J which satisfies the lemma. Standard estimates show that this holds when

$$m > 5\Delta^2 \log n. \qquad \square$$

To prove the theorem, fix a family $J = \{F_1, \cdots, F_n\}$ as in the lemma. Consider a vertex with label i whose neighbors' labels are $j_1, \cdots, j_d$ where $d \le \Delta$. Since

$$F_i \not\subseteq \bigcup_{\nu=1}^d F_{j_\nu},$$

there is a $1 \le x \le m$ with $x \in F_i \setminus (\bigcup_{\nu=1}^d F_{j_\nu})$. The color of this vertex is x. It is easily verified that this is a proper m-coloring of G. $\square$

The same proof yields, more generally, the following corollary.

COROLLARY 4.1. *Let G be a graph whose vertices are properly colored with k colors and whose largest degree is Δ. It is possible to color G with $5\Delta^2 \log k$ colors in one unit of time distributively.*

By iterating the coloring given by the corollary $\log^* n$ times, a $10\Delta^2 \log \Delta$-coloring is obtained. To eliminate the $\log \Delta$ term we need a result of the same type as Lemma 4.1, only in that range.

LEMMA 4.2. *Let q be a prime power. Then, there is a collection J of q^3 subsets of $\{1, \cdots, q^2\}$ such that if $F_0, \cdots, F_{[(q-1)/2]} \in J$ then $F_0 \not\subseteq \bigcup_1^{[(q-1)/2]} F_i$.*

Proof. Use Example 3.2 in [EFF] with $d = 2$. □

Select q to be the smallest prime power with $q \geq 2\Delta + 1$. There is certainly one with $4\Delta + 1 \geq q \geq 2\Delta + 1$. This construction transforms an $O(\Delta^3)$-coloring to an $O(\Delta^2)$-coloring as in the proof of Theorem 4.1.

Theorem 4.2 now follows.

THEOREM 4.2. *Let G be a labeled graph of order n with largest degree Δ. Then in time $O(\log^* n)$ it is possible to color G with $O(\Delta^2)$ colors in a distributive synchronous algorithm.* □

The previous proof is nonconstructive in that there is no known explicit construction as good as that which Lemma 4.1 guarantees. However, as Karloff pointed out [K], the explicit geometric construction in Example 3.2 of [EFF] enables us, in the same way, to reduce a k-coloring to an $O((\Delta^2 \log^2 k))$-coloring in one timestep. (The previous proof reduces a k-coloring to an $O((\Delta^2 \log k))$-coloring in a step.) The rest of the proof remains unchanged, yielding the same results with only slightly worse constants.

Proposition 3.4 of [EFF] sets a bound on the power of the present method, showing that it does not enable one to color with fewer than $\binom{\Delta+2}{2}$ colors. That proposition shows that set systems of the type that would allow further reduction of the number of colors do not exist. Other algorithms may still be capable of coloring with fewer colors. It would be interesting to decide whether this quadratic bound can be improved when time bounds rise from $O(\log^* n)$ to polylog, for instance.

5. Some final remarks. It is well known now how to find a maximal independent set in a graph by means of an NC algorithm ([KW], [ABI], [L]). The algorithm in [L] can even be viewed as a randomized distributed algorithm. However, an algorithm which is both deterministic and distributed still eludes us. In [GP] it is shown how to find an MIS in time $\log^* n$ for graphs of bounded degree. It is not clear what the situation is for unbounded degrees. In particular, can it always be found in polylogarithmic time in the present model? This, if true, would be a significant improvement over the above-mentioned studies.

A standard trick (e.g., [L]) allows us to transform an efficient MIS algorithm to one for $(\Delta + 1)$-coloring: Take the cartesian product of G with $K_{\Delta+1}$. It is easily verified that $(\Delta+1)$-colorings of G and MISs of $G \times K_{\Delta+1}$ are in a natural 1:1 correspondence. It is therefore particularly interesting to find out the best time complexity in terms of n for finding a $(\Delta + 1)$-coloring, and in particular whether polylogarithmic time suffices.

The fact that randomness does not help in coloring (Corollary 2.1) is thought-provoking: Getting a deterministic polylog-time algorithm for MIS seems hard, though simple randomized distributed algorithms are known. It is an intriguing problem to classify distributed graph problems according to how much randomization can help in solving them.

6. Acknowledgments. Comments recieved by a number of colleagues helped improve this paper in a significant way. A comment by Noga Alon helped simplifying the proof of Theorem 2.1. Howard Karloff's comment on the proof of Theorem 4.1 is recorded in the text, and a similar remark was made by Noga Alon as well. That randomization does not help came up in discussions with Dror Zernik. Helpful comments were also received from the referees. I am grateful to all of them.

REFERENCES

[A] B. AWEBUCH, Private communication.

[ABI] N. ALON, L. BABAI, AND A. ITAI, *A fast and simple randomized algorithm for the maximal independent set problem*, J. Algorithms, 7 (1986), pp. 567–583.

[B] B. BOLLOBAS, *Random Graphs*, Academic Press, New York, 1985.

[CV] R. COLE AND U. VISHKIN, *Deterministic coin tossing and accelerating cascades: micro and macro techniques for designing parallel algorithms*, in Proc. 18th ACM Symposium on Theory of Computing, 1986, pp. 206–219.

[EFF] P. ERDÖS, P. FRANKL AND Z. FÜREDI, *Families of finite sets in which no set is covered by the union of r others*, Israel J. Math., 51 (1985), pp. 79–89.

[GP] A. V. GOLDBERG AND S. A. PLOTKIN, *Efficient parallel algorithms for* $(\Delta + 1)$*-coloring and maximal independent set problems*, in Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 315–324.

[JS] R. E. JOHNSON AND F. B. SCHNEIDER, *Symmetry and similarity in distributed systems*, in Proc. 4th ACM Symposium on Principles of Distributed Computing, 1985, pp. 13–22.

[K] H. KARLOFF, Private communication.

[KSS] D. J. KLEITMAN, J. SHEARER AND D. STURTEVANT, *Intersections of k-element sets*, Combinatorica, 1 (1981), pp. 381–384.

[KW] R. M. KARP AND A. WIGDERSON, *A fast parallel algorithm for the maximal independent set problem*, in Proc. 16th ACM Symposium on Theory of Computing, 1984, pp. 266–272.

[L] M. LUBY, *A simple parallel algorithm for the maximal independent set problem*, in Proc. 17th ACM Symposium on Theory of Computing, 1985, pp. 1–10.

[LPS] A. LUBOTSKY, R. PHILLIPS AND P. SARNAK, *Ramanujan graphs*, Combinatorica, 8 (1988), pp. 261–278.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 203-215, April 1992

© 1992 Society for Industrial and Applied Mathematics
001

PARTITIONING PLANAR GRAPHS*

THANG NGUYEN BUI† AND ANDREW PECK‡

Abstract. A common problem in graph theory is that of dividing the vertices of a graph into two sets of prescribed size while cutting a minimum number of edges. In this paper this problem is considered as it is restricted to the class of planar graphs.

Let G be a planar graph on n vertices and $s \in [0, n]$ be given. An s-partition of G is a partition of the vertex set of G into sets of size s and $n-s$. An *optimal s-partition* is an s-partition that cuts the fewest number of edges. The main result of this paper is an algorithm that finds an optimal s-partition in time $O(b^2n^32^{4.5b})$, where b is the number of edges cut by an optimal s-partition. In particular, by letting $s = \lfloor n/2 \rfloor$ the immediate corollary that any planar graph with small ($O(\log n)$) bisection width may be bisected in polynomial time is obtained.

Furthermore, suppose that a planar embedding $\hat{G}$ of G is also given such that the embedding of each biconnected component in $\hat{G}$ is at most m-outerplanar (such an embedding is called *m-outerplane-separable*). An algorithm for finding optimal s-partition of G in time $O(m^2n^32^{3m})$ is also given.

Key words. graph algorithms, k-outerplanar graphs, graph bisection, graph partitioning, edge separators, planar graphs

AMS(MOS) subject classifications. 05C85, 68Q20, 68Q25, 68R10

1. Introduction. A common problem in graph theory is that of dividing the vertices of a graph into two sets of prescribed size while cutting a minimum number of edges. In this paper we consider this problem restricted to the class of planar graphs.

Let a graph G on n vertices and an integer $s \in [0, n]$ be given. An *s-partition* of G is a partition of the vertex set of G into two disjoint sets of size s and $n-s$. The *cutset* of an s-partition is the set of edges with endpoints in both parts of the partition, and the cardinality of the cutset is called the *cutsize* of the s-partition. Given a graph G and an integer s, the *graph partitioning problem* is the problem of finding an *optimal s-partition*, that is, one for which the cutsize is minimal over all s-partitions of the graph. When $s = \lfloor n/2 \rfloor$, this is called the *graph bisection* problem.

The problem of finding the minimum bisection of a graph arises in many graph algorithms which use the divide-and-conquer approach. A notable example is the VLSI placement and routing problem [3]. It is also observed that many other divide-and-conquer based algorithms run much faster on graphs in which a small bisection can be found, for example, in the area of sparse matrix computation [7]; see also [13]. Despite its wide application, there is no known efficient algorithm for the graph bisection problem, since for general graphs the bisection problem is NP-complete. In fact, there are no known polynomial time approximation algorithms. Consequently, work has centered on finding heuristics with good average case behaviour. A number of methods are employed including a hill climbing heuristic [10], simulated annealing [11], network flow [6], and an approach based on the eigenvalues of the adjacency matrix [5]. Only the last two approaches give provably good average case behaviour. Most of the other known heuristics have no theoretical analysis.

* Received by the editors February 27, 1989; accepted for publication (in revised form) May 2, 1991. A preliminary version of this paper appeared in the Proceedings of the 26th Annual Allerton Conference on Communication, Control and Computing, Alberton House, Monticello, Illinois, 1988, pp. 798-807.

† Computer Science Department, The Pennsylvania State University, University Park, Pennsylvania 16802.

‡ Present address, Room 4H-314, Bell Laboratories, Indian Hill, 2000 North Naperville Road, Naperville, Illinois 60566-7033.

For special classes of graphs the situation is a little bit better. For example, an $O(n^2)$ time algorithm exists for trees [8], [14]. However, little is known for the class of planar graphs, including whether the problem is NP-complete. A motivation for studying this problem on planar graphs is that many examples found in practice involve planar graphs, e.g., VLSI placement. Another motivation is the close relation between the graph bisection problem and the vertex separator problem in planar graphs for which numerous applications have been shown [13]. In [15] Rao considers similar problems, namely the edge separator problem and some of its variations on planar graphs. The results there, however, do not extend to the case of bisection since the sizes of the two subgraphs are allowed to be between $\frac{1}{3}$ and $\frac{2}{3}$ of the original graph.

To describe the results in this paper we need the concept of a *k-outerplanar* embedding of a planar graph. Intuitively, a k-outerplanar embedding is an embedding consisting of vertex-disjoint outerplane graphs nested at most k deep. (An *outerplane* graph is a planar-embedded graph such that all vertices lie on the exterior face.) If an embedding of a planar graph is such that each biconnected component of the embedding is at most m-outerplanar, then we call such an embedding *m-outerplane-separable.*

The main result of this paper is an algorithm for finding an optimal s-partition of a planar graph in time $O(b^2n^3 2^{4.5b})$, where b is the size of the optimal s-partition. As a corollary, we have a polynomial time algorithm for finding the optimal s-partition of a planar graph if the optimal s-partition is of size $O(\log n)$.

In addition, we show that if an n-vertex planar graph G and $s \in [0, n]$ be given together with an m-outerplane-separable planar embedding of G, then the optimal s-partition of G may be determined in time $O(m^2n^3 2^{3m})$. Thus if G has an $O(\log n)$-outerplane-separable embedding then we can find an optimal s-partition of G in polynomial time. Note that for a planar graph the existence of an $O(\log n)$-outerplane-separable embedding does not imply the existence of an $O(\log n)$ s-partition, and vice versa. Our algorithm for partitioning m-outerplane-separable graphs is based on a dynamic programming scheme first proposed in [2].

The remaining sections of the paper are as follows. Section 2 introduces necessary terminology. In § 3 we develop a table structure for representing optimal partitions of a graph. In § 4 we give algorithms for partitioning planar graphs that have k-outerplane and k-outerplane-separable embeddings. Section 5 describes algorithms for partitioning general planar graphs by reducing it to the problem of partitioning k-outerplane-separable embeddings of planar graphs. Conclusions are given in § 6.

2. Definitions and terminology. A *plane graph* is an embedding of a planar graph. In the remainder of this paper we will assume G to be a connected n-vertex plane graph. Since a linear time algorithm exists (see [9]) to find a planar embedding of a planar graph, the assumption that the input graph is plane does not affect our conclusions. Extension of our results to disconnected plane graphs is easy.

2.1. Vertex level, *k*-outerplanarity. Let G be an n-vertex plane-graph. All vertices on the exterior face of G are *level* 1 vertices. Remove all vertices of level less than or equal to i, together with their incident edges. The vertices on the new exterior face are the *level* $(i+1)$ vertices.

A planar embedding of a graph is *k-outerplanar* if it has no vertices of level greater than k. A planar graph is *k-outerplanar* if it has a k-outerplanar embedding. Every planar graph is k-outerplanar for some k. The terms *outerplanar* and 1-*outerplanar* are equivalent. In keeping with the familiar term plane graph we use *k-outerplane graph* to denote a k-outerplanar embedding of a planar graph.

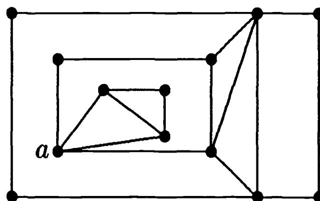


FIG. 2.1. *The above graph may be characterized as* 3*-outerplane,* 2*-outerplane-separable (by virtue of articulation point a), or* 2*-outerplanar (since a* 2*-outerplanar embedding exists). For the above embedding there are six vertices at level* 1*, four at level* 2*, and three at level* 3.

In [4] it is shown that given a planar graph G the minimum k can be found such that G is k-outerplanar and a k-outerplane embedding of G can be constructed. The algorithm given there runs in time $O(k^3n^2 \log n)$. Thus for the remainder of the paper we assume that if G is k-outerplanar then we are also given a k-outerplane embedding of G.

A *level edge* is an edge connecting two vertices at the same level in a k-outerplane graph. An *interlevel edge* connects a vertex at some level i to a level $(i-1)$ or level $(i+1)$ vertex. It is clear that a k-outerplane graph has only level and interlevel edges.

Consider a set of vertices in a k-outerplane graph which are connected by a path consisting only of level edges. A *level subgraph* is a maximal set of such vertices together with all level edges connecting them. A *level i subgraph* is a level subgraph containing level i vertices. If $i > 1$, then a level i subgraph is nested within and is connected via interlevel edges to exactly one level $(i-1)$ subgraph. Many level $(i+1)$ subgraphs may be nested within one level i subgraph.

An embedding of a planar graph is *k-outerplane-separable* if the biconnected components of the embedding are k-outerplane. As before we will use the term k-outerplane-separable graph to denote the k-outerplane-separable embedding. Intuitively, any nontrivial k-outerplane-separable graph looks like nested k-outerplane components each (except the outermost) sharing a single vertex with an enclosing level subgraph. (See Fig. 2.1.)

2.2. Vertex partitions. Recall the definitions of an s-partition, cutset, and cutsize in § 1. For a particular partition we will arbitrarily label the two vertex sets *left side* and *right side.* Thus, it will make sense to speak of a vertex as being, for example, "on the left side" of the partition. Once so labeled, a partition will be considered distinct from the partition obtained by exchanging the labels "left" and "right." The number of vertices on the left side of a partition is called the *partition size.*

A *constrained partition* is a partition formed subject to the requirement that particular vertices must appear on the left and certain others on the right. The set of constraining vertices is called the *constraint set.* Given a constraint set, we will speak of an *assignment* of its vertices to the left and right sides of a partition, this assignment being the actual constraint imposed on the partition. Size and optimality for a constrained partition have the same meaning as in the unconstrained case above with the exception that some partition sizes may be impossible.

3. Slice table. In this section we develop the table structure that will allow us to adapt a dynamic programming strategy from [2] to the partition problem on k-outerplanar graphs. The strategy is based on a decomposition of a graph into components called *slices.* We introduce the notion of a slice and show how to construct a table recording optimal partitions of a slice subject to constraints on the boundary

vertices of the slice. Finally, we describe a set of table operations that update a table to represent a slice which includes additional neighboring vertices or which is the merger of two adjacent slices.

For the sake of intuition it is useful to consider the case of simply nested graphs in this section. By a simply nested graph we mean a k-outerplane graph in which each level subgraph is a simple cycle and for each $i \in [1, k]$ there is exactly one level i subgraph. In this case it is easy and possible to describe the process of decomposing the graph into slices separately from the process of computing the tables for the slices and the merging of these tables. For the case of general plane graphs, the decomposing process is accomplished by a more complicated scheme and is intertwined with the process of computing and merging of the tables. This process is described in [2], and the reader is referred to that paper for more details. The concepts described in §§ 3.2 and 3.3 remain the same in either simply nested or general plane graphs.

3.1. Slice boundary and slice. A *level i slice boundary*, or simply *slice boundary* when i is understood, is a set of i vertices, one from each of levels 1 to i, such that a line joining the vertices in order of level number does not intersect any edge which does not have an endpoint on the boundary.

A *slicing* of a graph is a selection of slice boundaries such that

(i) every level i vertex is part of a level i slice boundary;

(ii) every level i slice boundary, $i > 1$, is the extension of some level $(i-1)$ slice boundary;

(iii) a line drawn through the vertices of one boundary in order of level number does not cut a similar line drawn for any other boundary (two boundaries may coincide at points).

Given a slicing of a simply nested k-outerplane graph, a *level i slice* is a subgraph subtended by two (not necessarily distinct) level i slice boundaries. More specifically, if one boundary is chosen as the first boundary, the slice contains the vertices of the boundaries and all vertices encountered in counterclockwise traversals of level subgraphs beginning at the first boundary and ending at the second boundary. All edges with both endpoints in the slice are included except any that leave the first boundary in a clockwise direction. Two slices are *adjacent* if they share a common boundary. In this case, we modify the definition of slice so that any interlevel edge with both endpoints in a common boundary is arbitrarily included in one slice or the other, but not both.

One of the contributions of [2] is to show how to extend the notion of a slice to the case of general planar graphs while preserving an essential property that in a slicing of a k-outerplane graph adjacent slices share no more than k vertices. This property is essential for bounding the running time of the dynamic programming strategy.

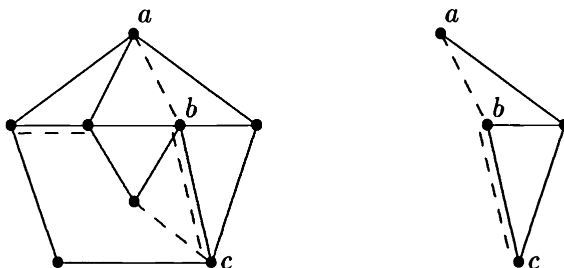
3.2. Partition vector. A *partition vector* for a graph G on m vertices is an $(m+1)$-vector P, where $P[i]$, $i \in [0, m]$, is the cutsize of an optimal i-partition of G. Two elements, $P[0]$ and $P[m]$, representing partitions with all vertices on one side or the other are always zero and are included for computational convenience.

Let A and B be two sets of k vertices of G. Designate the vertices in $A \cup B$ as a constraint set for partitions of G. Let X be a k-bit vector such that $X[i] = 1$ $(X[i] = 0)$ if the ith element of A is constrained to be on the left (right) side of the partition. Let a k-bit vector Y be defined similarly for B. A *constrained partition vector P_{XY}* is defined as in the unconstrained case except that $P_{XY}[i]$ is now the optimal i-partition obtainable subject to the constraints represented by X and Y. If a constrained i-partition is impossible, then let $P_{XY}[i] = \infty$. Since A and B are not required to be disjoint, the

values of X and Y may represent inconsistent assignments of some vertex in $A \cap B$. In this case let $P_{XY}[i] = \infty$ for all i, that is, the table entry is undefined for all i.

In the above definition of a constrained partition vector replace G by a slice S of G. Let the constraint sets A and B be the level 1 to level j, $j \leqq i$, vertices of the slice boundaries of S. Then X and Y are both j-bit vectors and P_{XY} is a constrained partition vector for S. A *slice table* for S is an arrangement of the 2^{2j} possible constrained partition vectors for S in a square table indexed by X and Y. That is, if X_B, Y_B are the binary values represented by the bit vectors X and Y, then the (X_B, Y_B)th table entry is the constrained partition vector P_{XY}. Note that half the information in the table is redundant since $P_{XY}[s] = P_{\bar{X}\bar{Y}}[n-s]$ where $\bar{X}$, $\bar{Y}$ are the bit complements of X, Y. (See Fig. 3.1.) In the special case of a constraint set consisting of a single vertex v it will be convenient to refer to a constrained partition vector P_v which, because of redundancy, contains all the information of a $2^1 \times 2^1$ table.

3.3. Table operations. We state three lemmas describing operations on slice tables—*merger*, *extension*, and *constraint removal.* We prove the lemma for the table merger operation in detail and sketch proofs for the remaining two which are similar.



	00	01	10	11
00	$(0, 3, -, -, -)$	$(-, -, -, -, -)$	$(-, 2, 3, -, -)$	$(-, -, -, -, -)$
01	$(-, -, -, -, -)$	$(-, 2, 3, -, -)$	$(-, -, -, -, -)$	$(-, -, 2, 1, -)$
10	$(-, 1, 2, -, -)$	$(-, -, -, -, -)$	$(-, -, 3, 2, -)$	$(-, -, -, -, -)$
11	$(-, -, -, -, -)$	$(-, -, 3, 2, -)$	$(-, -, -, -, -)$	$(-, -, -, 3, 0)$

FIG. 3.1. *A 2-outerplane graph with some possible slice boundaries indicated by dashed lines. A partition vector for the entire graph is* $P = (0, 2, 3, 4, 4, 4, 3, 2, 0)$. *Above is a slice table for the slice shown on the right with boundaries* (a, b) *and* (c, b). *The rows are indexed with the constraints on the set* $\{a, b\}$ *and the columns are indexed with the constraints on the set* $\{c, b\}$. *Dashes in the table represent values of* ∞.

LEMMA 3.1 (Table merger). *Let S_1 and S_2 be adjacent level i slices for which slice tables have been computed. A slice table for the slice $S = S_1 \cup S_2$ may be computed from the tables for S_1 and S_2 in time $O(2^{3i} m^2)$ where m is the number of vertices in S.*

Proof. Let Z be the bit vector representing the left/right assignments of the vertices in the common boundary of S_1 and S_2. Let X, Y be the bit vectors reflecting left/right assignments of the vertices on the remaining boundaries of S_1 and S_2, respectively. In other words, X and Y are the constraints for the boundaries of S. Denote by P^1_{XZ},

P^2_{ZY}, and P_{XY} constrained partition vectors for S_1, S_2, and S, respectively. Denote by z the number of 1's in the bit vector Z. Thus, given a partition of the vertices of S, z is the number of vertices of the common boundary of the two subslices S_1 and S_2 which lie on the left side of the partition.

The table for S will have rows indexed by X and columns indexed by Y. It will contain the same number, 2^{2^i}, of entries as the tables for S_1 and S_2. Table entries will be $(m+1)$-vectors.

To see that the tables for S_1 and S_2 contain sufficient information to construct a table for S observe that an optimal constrained partition of S induces optimal constrained partitions of S_1 and S_2. Further, if a partition of S_1 and a partition of S_2 have the same left/right assignments of the common boundary vertices, then they combine to form a partition of S. The following two claims, which are straightforward to prove, formalize the argument. Let the number of vertices in S_1 and S_2 be m_1 and m_2, respectively, and let X, Y, Z, and z be as above.

CLAIM 3.1. *Let $x_1 \in [0, m_1]$, $x_2 \in [0, m_2]$, and $x \in [0, m]$ be such that $x_1 + x_2 - z = x$; then*

$$P^1_{XZ}[x_1] + P^2_{ZY}[x_2] \geqq P_{XY}[x].$$

Proof. The proof is simple. Note that if partitions of S_1 and S_2 have the same left/right assignment of the common boundary vertices, then the two partitions must together induce a partition of S. The size of the induced partition is the sum of the sizes of the partitions of S_1 and S_2 minus z the number of common vertices which are on the left side. The cutsize of the induced partition is the sum of the cutsizes of the partitions of S_1 and S_2 since the slices are edge disjoint. This cutsize cannot be smaller than the optimal cutsize $P_{XY}[x]$. □

CLAIM 3.2. *Given x, there exist $x_1 \in [0, m_1]$, $x_2 \in [0, m_2]$ and a bit vector Z such that $x_1 + x_2 - z = x$ and $P^1_{XZ}[x_1] + P^2_{ZY}[x_2] = P_{XY}[x]$.*

Proof. The proof of Claim 3.2 is also straightforward. Let p be an x-partition of S satisfying the constraints X and Y and having cutsize $P_{XY}[x]$. Let Z reflect the left/right assignment of the common boundary of S_1 and S_2 induced by p; then p induces partitions of S_1 and S_2, satisfying the constraints XZ and ZY, respectively. Let x_1 and x_2 be the sizes of these partitions. It is clear that $x_1 + x_2 - z = x$ and that the cutsize of p, namely, $P_{XY}[x]$, is equal to the sum of the cutsizes of the induced partitions. This sum, however, is greater than or equal to $P^1_{XZ}[x_1] + P^2_{ZY}[x_2]$, which is greater than or equal to $P_{XY}[x]$ by Claim 3.1. Hence we have $P_{XY}[x] = P^1_{XZ}[x_1] + P^2_{ZY}[x_2]$. □

Claims 3.1 and 3.2 imply that in order to determine $P_{XY}[x]$ from the tables for S_1 and S_2 we need only find values for Z, x_1, x_2 that minimize $P^1_{XZ}[x_1] + P^2_{ZY}[x_2]$ and such that $x = x_1 + x_2 - z$. Thus, to compute a table representing the merger of the slices S_1 and S_2 it is only necessary to scan each entry P^1_{XZ} once for each entry P^2_{ZY}. If m_1, m_2 are the numbers of vertices in S_1 and S_2, then this can be done in time proportional to $2^{3^i} m_1 * m_2$, which is $O(2^{3^i} m^2)$. □

LEMMA 3.2 (Table extension). *Let S be a level i slice containing m vertices for which a slice table has been computed. Let v be a level $(i+1)$ vertex such that the boundaries of S may be extended to include v and such that there is no edge from v to a level i vertex in S unless it is in one of the boundaries. A table for the extended slice may be computed from the table for S in time $O(2^{2^i} m)$.*

Proof (Sketch). The new level $(i+1)$ slice contains $m+1$ vertices. Thus, a slice table for the new slice will contain $2^{2i+2}(m+2)$-vectors. Since v appears twice in the

constraint set (in both slice boundaries) half of the new table entries, those representing conflicting assignments of v, will be undefined. The remaining table elements may be filled in from the table for S by making adjustments to reflect edges cut by the assignments of v. The only edges that need be considered are the (at most) two edges from v to level i vertices of S. An argument for correctness of the operation follows the same lines as for the table merger operation. Since the procedure requires only a single scan of the old and new tables, time is $O(2^{2^i}m)$. □

LEMMA 3.3 (Constraint removal). *Let S be a level i slice containing m vertices for which a $2^i \times 2^i$ slice table has been computed. A $2^j \times 2^j$ slice table, $j < i$, representing the same slice constrained only by the first j vertices of each border, may be computed in time $O(2^{2^i}m)$.*

Proof (Sketch). In the special case of a slice with a constraint set of a single vertex v we have a single constrained partition vector P_v. Entries in an unconstrained partition vector are $P[s] = \min(P_v[s], P_v[n-s+1])$. For nontrivial tables proceed as follows to compute an element of the new table. Group rows of the old table with indices that are identical except for the last $i-j$ bits. Likewise group the columns. The intersection of a row group with a column group contains $2^{2(i-j)}$ partition vectors. Compute a new partition vector, the sth element of which is the minimum of the sth elements of the $2^{2(i-j)}$ old vectors. The row and column indices of this new vector in the new table are the first j bits of the indices in the old table. If $j = 0$, then the resulting table is a single unconstrained partition vector representing optimal partitions of the slice. The process of constraint removal may be accomplished in one pass through the table or in time $O(2^{2^i}m)$. □

4. Partitioning k-outerplane graphs. In this section we show how to partition k-outerplane and k-outerplane-separable graphs in polynomial time. We also show that the methods extend to graphs with weighted edges. Remember that in § 2 we defined a k-outerplane graph to be an embedding of a k-outerplanar graph. Since a polynomial time algorithm exists for finding a k-outerplane embedding of a k-outerplanar graph [4], our assumption that the appropriate-embedding is given does not affect our result. This remark, of course, also applies to k-outerplane-separable graphs.

In [2] Baker provides a general method for building tables of solutions for various combinatorial optimization problems on k-outerplane graphs. We provide a brief description of Baker's method and refer the reader to the original paper for details.

Baker's method works generally with min-/maximization problems on the vertices of k-outerplane graphs in which the quantity to be optimized is such that a solution for the entire graph may be computed from constrained optimal solutions on subgraphs which are disjoint but for boundary vertices. A problem-dependent table structure records optimal constrained solutions on slices of a graph. A problem independent scheme then builds up a solution table for the entire graph. The scheme works by first defining trivial slices for which tables are easily computed and then extending and merging these slices (while carrying out the corresponding operations on the associated tables) until a single "slice" and table represents the entire graph.

To gain an intuition for the working of Baker's method consider a level i slice S in a slicing of some graph. If $i = 1$, then S consists of either a single vertex or a pair of vertices connected by an edge so a slice table is easily constructed. If $i > 1$, then S contains at most two level i vertices, say u and v, as well as the union of one or more level $(i-1)$ slices. If slice tables already exist for the level $(i-1)$ slices, then a table for S may be constructed through a series of extension and merger operations (see § 3). Care is taken to avoid violating planarity in choosing the vertex u or v for each

table extension and at the appropriate stage the effect of the edge (u, v) on solutions is incorporated in the new table.

Using the procedure just described it is easy to construct a recursive algorithm to yield tables for a slicing of a simply nested graph. The resulting tables may then be merged and constraints removed to obtain an unconstrained solution for the entire graph. At all times the table sizes are bounded by a factor of 2^{2k} and the number of tables and hence the number of table operations is bounded by the size of the graph.

Baker extends the above method to planar graphs that are not simply nested. A systematic method for defining slices is developed which handles the cases of articulation points and chords in level subgraphs while assuring that each slice shares boundaries no longer than k with neighboring slices. Further, the algorithm for drawing the slices provides a natural order for the application of extension, merger, and constraint removal operations which assures that these operations occur a number of times linear in the size of the graph. Thus, the time required is linear in the number of tables multiplied by the time for the most costly table operation.

For the graph partition problem the most costly operation is table merger, requiring time $O(2^{3^i}m^2)$, where m is the number of vertices in the combined slices and i is the maximum level of the slice. Since $m \leqq n$ and $i \leqq k$, the following theorem is an immediate result of the above discussion and the operations defined in § 3.3.

THEOREM 4.1 (Partition of k-outerplane graphs). *Let G be an n-vertex k-outerplane graph and $s \in [0, n]$ be given. An optimal s-partition of G may be found in time $O(2^{3k}n^3)$.*

The algorithm returns a single partition vector P for the entire graph, where $P[s]$ is the cutsize of an optimal s-partition of the graph. If the actual partition is required, then it may be found by preserving the tables and retracing the steps of the computation. Space required by the tables is that for $2^{2k}n$ partition vectors which is $O(2^{2k}n^2)$.

The above scheme may be made to return a constrained partition vector P_v representing partitions constrained by a single vertex v on the exterior face (at the start select the appropriate vertex and at the conclusion remove all constraints except the constraint on this vertex). This is important to the following theorem, which follows from the observation that biconnected components may be processed separately to yield tables constrained only by articulation points.

THEOREM 4.2 (Partition of k-outerplane-separable graphs). *Theorem 4.1 holds for G an n-vertex k-outerplane-separable graph.*

Proof (Sketch). Proof is by construction of a simple recursive algorithm to process k-outerplane components. Note that during processing every vertex appears at some time in the constraint set. When an articulation point becomes a constraint vertex, recursively process each previously unprocessed component of the graph that shares that articulation point. Merge the resulting partition vectors into the table and continue. The table merger procedure need only be altered to reflect the possible disparity in sizes of the tables to be merged, i.e., if the articulation point occurs at level i, then each partition vector (the equivalent of a 2×2 table) must be merged into a $2^i \times 2^i$ table.

The need to identify biconnected components is not a factor in the time bound since linear time algorithms exist to do this. No other significant modifications of Theorem 4.1 are needed. Therefore, running time remains linear in the cost of table mergers. □

For graphs with weighted edges *cutsize* and *optimal cutsize* may be redefined in terms of the total weight of edges in the cutset. To modify the algorithm to accommodate such graphs we need only change procedures in which the effect of cutting an individual edge is considered as is the case, for example, in table extension. The cost of cutting an edge is now simply its weight rather than 1. Running time is unaffected. The space

requirement is unaffected so long as we accept a word model in which individual elements in a partition vector may have arbitrary size. The following corollary is the result of these observations.

COROLLARY 4.3. *Theorems* 4.1 *and* 4.2 *hold for graphs with weighted edges.*

5. Partitioning planar graphs with small optimal cutsize. In § 4 we showed how to find an optimal s-partition of an n-vertex plane graph G having m-outerplane biconnected components. We obtained a polynomial time algorithm when $m = O(\log n)$. In this section we show that we can still get a polynomial time algorithm when m is not $O(\log n)$ provided that the cutsize of the optimal s-partition is small. Note that a small cutsize does not imply that m is small, or vice versa.

The main idea here is to construct a series of edge-weighted k-outerplane-separable subgraphs G_i^* of a plane graph G that isolate every cutset no larger than k of G. By applying this construction together with Theorem 4.2 for increasing values of k, we obtain an optimal partition of the graph.

5.1. Isolating optimal cutsets. To isolate a cutset with cutsize k or less we construct a set of weighted subgraphs G_i^*, $i \in [0, k]$ with the property that an edge set S, $|S| \leqq k$, is the cutset of an optimal s-partition of G if and only if it is the cutset of an optimal s-partition of at least one of the subgraphs. The main idea of the construction is as follows. For any level $(j+2)$ subgraph, $j = i \bmod (3k+3)$, remove all but one of the edges connecting it to the enclosing level $(j+1)$ subgraph. In addition, give weight ∞ to all edges with endpoints in a level $(j+1)$ or $(j+2)$ subgraph, and weight 1 to the rest. The result is a weighted $(3k+3)$-outerplane-separable graph G_i^*. We will show that if an optimal s-partition of G has cutsize no greater than k, then for any optimal s-partition of G there is some l such that the cutset is contained completely among the weight 1 edges of G_l^*. Furthermore, it is straightforward to show that the same set of edges is also the cutset for an optimal s-partition of G_l^*.

LEMMA 5.1. *Let G be an m-outerplane graph and $s \in [0, n]$. If an optimal s-partition of G has cutsize $w \leqq k$; then an edge-weighted subgraph G^* of G may be constructed such that G^* is $(3k+3)$-outerplane-separable and such that an optimal s-partition of G^* is an optimal s-partition of G.*

Proof. If $k \geqq m/3 - 1$, then G is $(3k+3)$-outerplane and we are done. Assume that $k < m/3 - 1$.

Let S be the cutset for some optimal s-partition of G. By assumption there are no more than k edges in S. For each $i \in [0, k]$ define J_i, G_i, H_i as follows.

- $J_i = \{j \mid j = 3i \bmod (3k+3), j \in [0, m-2]\}$.
- G_i is the subgraph consisting of G minus all level $(j+1)$ and $(j+2)$ subgraphs and their incident edges for all $j \in J_i$.
- H_i is the set of all edges of G which are not in G_i.

For each $i \in [0, k]$ construct a weighted subgraph G_i^* as follows. Let $j \in J_i$. For every level $(j+2)$ subgraph of G remove all but one interlevel edge connecting it to the enclosing level $(j+1)$ subgraph. Give weight ∞ to all edges of G_i^* that are in the set H_i. Give weight 1 to all other edges. The result is an edge-weighted $(3k+3)$-outerplane-separable graph. Call the edges with weight ∞ *heavy* edges and the remaining edges *light* edges. Observe that the light edges are exactly the edges of G_i and that only heavy edges have endpoints in the level $(j+1)$ and $(j+2)$ subgraphs.

The $k+1$ edge sets H_i are clearly disjoint. Thus, there is some $l \in [0, k]$ such that H_l contains no edges of S. It follows that all of the edges of S are in G_l. By our previous observation all the edges of S are light edges in G_l^*.

The following claims will complete the proof of Lemma 5.1.

CLAIM 5.1. *A partition of the vertices of G which separates the endpoints of an edge which is not in G_l^* will also separate the endpoints of some heavy edge in G_l^*.*

An edge that is not in G_l^* must connect a level $(j+1)$ and a level $(j+2)$ subgraph, $j \in J_l$. In G_l^* any two such level subgraphs contain only heavy edges and are joined by a single heavy edge. Thus there is a path in G_l^* joining the endpoints of a and consisting entirely of heavy edges. Any partition of the vertices of G and, hence, of G_l^* which separates the endpoints of a must also cut some edge on this path.

CLAIM 5.2. *Any cutset of G_l^* that includes only light edges is also a cutset for G.*

If this is not the case, then it must be that the vertex partition induced by the cutset on G_l^* separates the endpoints of some edge $a \in G - G_l^*$. Together with the first claim this produces a contradiction.

CLAIM 5.3. *Any cutset of weight less than ∞ of G_l^* is also a cutset of G.*

This follows from Claim 5.2 since such a cutset can contain only light edges.

CLAIM 5.4. *An optimal s-partition of any G_l^* has cutsize no less than w.*

Immediate from Claim 5.3 since a cutsize less than w would imply an optimal s-partition of G with cutsize less than w.

CLAIM 5.5. *S is an optimal cutset for G_l^*, for some $l \in [0, k]$.*

Recall that S is the cutset for some optimal s-partition of G. Since $|S| = w \leqq k$, the edge set S is contained among the light edges of G_l^* for some $l \in [0, k]$. From Claim 5.4 S must be an optimal cutset for G_l^*.

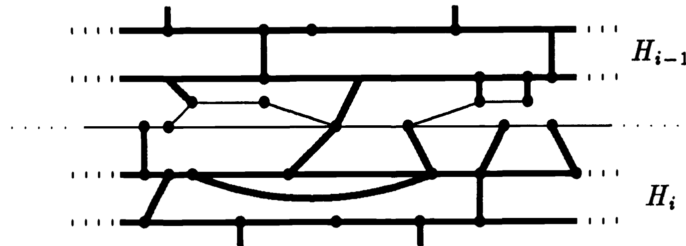


FIG. 5.1. *A section of a biconnected m-outerplane graph. Horizontal edges are in levels $(j-2)$ to $(j+2)$ for some $j \in J_i$. Thin edges are part of a level j subgraph. Thick edges are from the edge sets H_{i-1} (top) and H_i (bottom).*

To obtain the G^* of Lemma 5.1 we need only construct the $k+1$ graphs G_l^* and take as G^* that G_l^* with an s-partition of minimal cutsize. □

In the case where G is biconnected, a more careful analysis in the proof of Lemma 5.1 shows that we need to construct only half as many subgraphs in order to isolate an optimal cutset. Moreover, the subgraphs are only $(1.5k+1.5)$-outerplane-separable. This is demonstrated in the proof of the following lemma.

LEMMA 5.2. *If G is biconnected, then Lemma 5.1 holds with G^* a $(1.5k+1.5)$-outerplane-separable graph.*

Proof. If $k \geqq 2m/3-1$, then G is $(1.5k+1.5)$-outerplane and we are done. Assume that $k < 2m/3-1$. For each $i \in [0, \lfloor k/2 \rfloor]$ we define G_i and H_i as before and modify the definition of J_i:

$$J_i = \{j \mid j = 3i \bmod (1.5k+1.5), j \in [0, m-2]\}.$$

As before let S be the cutset of an optimal s-partition of G, $|S| \leqq k$. We must ensure that there is some $l \in [0, \lfloor k/2 \rfloor]$ such that H_l contains no edges of S. (See Fig. 5.1.)

For increasing values of i examine each H_i in turn. We will show that each H_i either contains no edge of S or *accounts* for at least two edges of S. If H_i contains no

edges of S, then we are done. If H_i contains two or more edges of S, then let H_i account for just these edges. Otherwise, H_i contains exactly one edge $e \in S$. Observe that e cannot lie on any simple cycle consisting only of edges of H_i since this would imply another edge of S in H_i. Thus, e is an interlevel edge with one endpoint in a level j subgraph F, where either $j \in J_i$ or $j \in J_{i+1}$. In either case biconnectedness implies that e is on a simple cycle consisting only of edges from H_i and F. There must be an edge $e' \in S$ among the edges of F in this simple cycle. If $j \in J_{i+1}$, then let H_i account for the two edges e and e'. If $j \in J_i$, then e' may already have been accounted for by H_{i-1} if $i>0$. However, note that any edge in H_i is enclosed within some face of F which implies that e' is on a cycle of edges from F. Thus, F must contain at least another edge in S different from e'. Let H_i account for this edge and e.

From the above discussion for each i the edge set H_i accounts for two or more edges of S if it accounts for any, and no edge is accounted for by more than one set H_i. Thus we must find an $l \leqq \lfloor k/2 \rfloor$ such that H_l accounts for no edges of S. Consequently, H_l contains no edges of S. The remainder of the proof proceeds as in Lemma 5.1 □

5.2. The algorithms. We describe two algorithms that use the construction of Lemma 5.2. Algorithm 5.1 partitions an m-outerplane biconnected graph and is easily extended to do the same for an m-outerplane-separable graph. Algorithm 5.2 uses Algorithm 5.1 to partition a general plane graph and gives us the proof of the main theorem of the paper.

Let $G=(V, E)$ be m-outerplane and biconnected and let $k \leqq |E|$. The following algorithm finds an optimal s-partition of G for all s such that the cutsize of the optimal s-partition is no more than k.

ALGORITHM 5.1

Input:

$G=(V, E)$ biconnected and m-outerplane;
$k \leqq |E|$;
v a vertex on the external face of G.

Output:

A vector P_v.

Step 1. If $k \geqq 2m/3-1$ then apply Theorem 4.1 returning a partition vector P_v for the entire graph; otherwise do Steps 2–4.

Step 2: Create $\lfloor k/2 \rfloor + 1$ edge-weighted $(1.5k+1.5)$-outerplane-separable graphs G_i^* as described in Lemma 5.2.

Step 3: Use Theorem 4.2 to obtain a partition vector P_v^i for each G_i^*.

Step 4: Return a vector P_v in which $P_v[s] = \min_{i \in [0, \lfloor k/2 \rfloor]} P_v^i[s]$.

Observe that if steps 2–4 are executed, then P_v is not strictly a partition vector for G since $P_v[s]$ is guaranteed to be optimal only if it is no more than k. If $P_v[s] > k$, then the optimal s-partition has cutsize greater than k, but has not necessarily been found.

The correctness of Algorithm 5.1 follows from consideration of Theorems 4.1 and 4.2, Corollary 4.3, and Lemma 5.2. Assuming an appropriate data structure for the input graph G, the construction of G_i^* proceeds in linear time. Running time for steps 2–4 is therefore dominated by step 3, which must be repeated $\lfloor k/2 \rfloor + 1$ times. By Theorems 4.2 and 4.3, the total time requirement is $\min(O(2^{3m}n^3), O(k2^{4.5k}n^3))$.

Algorithm 5.1 can be extended without increasing the running time to handle input graphs which are m-outerplane-separable. The extension is analogous to Theorem 4.2. As in Theorem 4.2, the constraint on v may be removed. We will refer below to

the *extended* Algorithm 5.1. We are now ready to give an algorithm to partition general planar graphs.

ALGORITHM 5.2

Input:

$G=(V, E)$, an n-vertex planar graph;
s, the size of the partition sought.

Output:

An optimal s-partition of G.

Step 1. Obtain a planar embedding of G, determine m such that this embedding is m-outerplane-separable.

Step 2. For increasing values of k apply the extended Algorithm 5.1 with input G, k, returning vector P until $P[s]=k$ or $k=2m/3-1$;

Step 3. If step 2 does not return an optimal s-partition, then apply Theorem 4.2 to the full graph G to find an optimal s-partition.

If an optimal s-partition of G has cutsize $b<2m/3$, then step 2 repeats b times. The running time for the ith repetition of step 2 is $O(i2^{4.5i}n^3)$. If step 3 is not executed, then the final cost is $O(b^2 2^{4.5b} n^3)$. If step 3 is executed, then it must be that $b \geqq 2m/3-1$. In this case step 3 finds an optimal s-partition in time $O(2^{3m}n^3)$ by application of Theorem 4.2. However, since we must account for the $2m/3-1$ unsuccessful iterations the final running time is $O(m^2 2^{3m} n^3)$. Algorithm 5.2 gives us the main theorem of the paper.

THEOREM 5.1. *Let an n-vertex planar graph G and* $s \in [0, n]$ *be given together with a planar embedding of G. Let m be such that this embedding is m-outerplane-separable and let b be the cutsize of an optimal s-partition of G. Then the optimal s-partition of G may be determined in time* $\min(O(b^2 n^3 2^{4.5b}), O(m^2 n^3 2^{3m}))$.

Corollary 5.1 follows from Theorem 5.1 by substituting $c \log n$ for b.

COROLLARY 5.1. *Let G be an n-vertex planar graph and* $s \in [0, n]$ *be given. If the cutsize of an optimal s-partition is* $O(\log n)$ *or if an* $O(\log n)$*-outerplane-separable embedding of G is available, then an optimal s-partition of G may be found in polynomial time.*

Corollary 5.2 is a result of the observation that a bisection of a graph is an $\lfloor n/2 \rfloor$-partition or an $\lceil n/2 \rceil$-partition.

COROLLARY 5.2. *Let G be an n-vertex planar graph. If G has* $O(\log n)$ *bisection width or if an* $O(\log n)$*-outerplane-separable embedding of G is available, then an optimal bisection of G may be found in polynomial time.*

6. Conclusions. In this paper we showed that an optimal s-partition of a planar graph may be found in polynomial time if the cutsize of the optimal partition is $O(\log n)$ or if an embedding of the graph is available in which the embedding of each biconnected component is $O(\log n)$-outerplanar. Letting $s=\lfloor n/2 \rfloor$ or $s=\lceil n/2 \rceil$ we have the same result for the graph bisection problem.

In the first part of the paper we showed how to construct tables that adapt a dynamic programming strategy presented in [2] to the problem of partitioning a k-outerplanar embedded graph with unweighted edges. We then extended this result to graphs with weighted edges and to k-outerplane-separable graphs.

In the second half of the paper we showed how to construct an edge-weighted $(1.5k+1.5)$-outerplane-separable subgraph of a plane graph such that if an optimal s-partition of the subgraph has cutsize no more than k, then an optimal s-partition of the subgraph is an optimal s-partition of the parent graph. By constructing and

partitioning a sufficient number of these subgraphs we were able to find the optimal partition of the original graph.

Our method suggests some avenues for further study:

(1) It would be useful to find a different method of constructing the subgraphs of § 5.1 using a k smaller than b while preserving the essential property that some optimal cutset is guaranteed to be contained in some subgraph. In particular, if k can be held at most logarithmic in the size of the graph for some subclass of planar graphs, then a polynomial time algorithm for finding optimal s-partitions would result for that subclass.

(2) It seems that the running time of our algorithms can be improved by considering the order of processing the slices. This is because for some graph embeddings the processing order will affect the total running time of the table merger operations.

(3) In [1] certain results on planar graphs are extended to graphs with an excluded minor; it would be interesting to show that the results in this paper can be extended similarly.

REFERENCES

[1] N. ALON, P. SEYMOUR, AND R. THOMAS, *A separator theorem for graphs with an excluded minor and its applications*, Proc. Symposium on Theory of Computing, 1990, pp. 293–299.

[2] B. BAKER, *Approximation algorithms for NP-complete problems on planar graphs*, Proc. Symposium on Foundation of Computer Science, 1983, pp. 265–273.

[3] S. N. BHATT AND F. T. LEIGHTON, *A framework for solving* VLSI *graph layout problems*, Com. System Sci., 28 (1984).

[4] D. BIENSTOCK AND C. L. MONMA, *On the complexity of embedding planar graphs to minimize certain distance measures*, Algorithmica, to appear.

[5] R. BOPPANA, *Eigenvalues and graph bisection: An average-case analysis*, Proc. Symposium on Foundation of Computer Science, 1987, pp. 280–285.

[6] T. BUI, S. CHAUDHURI, T. LEIGHTON, AND M. SIPSER, *Graph bisection algorithms with good average case behavior*, Combinatorica, 7 (1987), pp. 171–191.

[7] J. R. GILBERT AND E. ZMIJEWSKI, *A parallel graph partitioning algorithm for a message-passing multiprocessor*, Cornell Computer Science Tech. Report, TR 87-803, Ithaca, NY, January 1987.

[8] M. GOLDBERG AND Z. MILLER, *A parallel algorithm for bisection width in trees*, unpublished notes, 1986.

[9] J. E. HOPCROFT AND R. E. TARJAN, *Efficient planarity testing*, J. Assoc. Comput. Mach., 21 (1974), pp. 549–568.

[10] B. W. KERNIGHAN AND S. LIN, *An efficient heuristic procedure for partitioning graphs*, Bell System Tech. J., 49 (1970), pp. 291–307.

[11] S. KIRKPATRICK, C. D. GELATT, JR., AND M. P. VECCHI, *Optimization by simulated annealing*, Science, 220 (1983), pp. 671–680.

[12] R. J. LIPTON AND R. E. TARJAN, *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36 (1979), pp. 177–189.

[13] ———, *Applications of a planar separator theorem*, SIAM J. Comput., 9 (1980), pp. 615–627.

[14] R. M. MACGREGOR, *On partitioning a graph: A theoretical and empirical study*, Ph.D. thesis, University of California, Berkeley, CA, 1978.

[15] S. RAO, *Finding near optimal separators in planar graphs*, Proc. Symposium on Foundation of Computer Science, 1987, pp. 225–237.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 216-227, April 1992
© 1992 Society for Industrial and Applied Mathematics
002

A POLYNOMIAL-TIME ALGORITHM FOR THE EQUIVALENCE OF PROBABILISTIC AUTOMATA*

WEN-GUEY TZENG†

Abstract. Two probabilistic automata are equivalent if for any string x, the two automata accept x with equal probability. This paper presents an $O((n_1+n_2)^4)$ algorithm for determining whether two probabilistic automata U_1 and U_2 are equivalent, where n_1 and n_2 are the number of states in U_1 and U_2, respectively. This improves the best previous result, which showed that the problem was in *coNP*.

The existence of this algorithm implies that the covering and equivalence problems for uninitiated probabilistic automata are also polynomial-time solvable. The algorithm used to determine the equivalence of probabilistic automata can also solve the path equivalence problem for nondeterministic finite automata without λ-transitions and the equivalence problem for unambiguous finite automata in polynomial time.

This paper studies the approximate equivalence (or δ-equivalence) problem for probabilistic automata. An algorithm for the approximate equivalence problem for positive probabilistic automata is given.

Key words. probabilistic automata, nondeterministic finite automata, unambiguous finite automata, equivalence, approximate equivalence, path equivalence

AMS(MOS) subject classifications. 68C25, 68D25

1. Introduction. A probabilistic automaton is a finite state machine with probabilistic transitions among states. For each string x, a probabilistic automaton has a certain probability of accepting x. Two probabilistic automata are *equivalent* if for any string x the two automata accept x with equal probability. While the equivalence problem for probabilistic automata was known to be in *coNP* [7], the question of whether the equivalence problem was polynomial-time solvable was left open. In this paper we present a polynomial-time algorithm for this problem. Furthermore, if two probabilistic automata are not equivalent then we demonstrate that our algorithm will output in polynomial time the lexicographically minimum string which the two automata will accept with different probabilities. As a consequence, the covering and equivalence problems for uninitiated probabilistic automata are also polynomial-time solvable.

For a probabilistic automaton U, the *state distribution* induced by a string x is the vector of probabilities that U ends up in each state when the input of U is string x. Our approach to the problem makes use of an algorithm which finds a basis for the vector space generated by the state distributions induced by all strings.

The technique we use to solve the equivalence problem for probabilistic automata in polynomial time has an interesting application to the path equivalence problem for nondeterministic finite automata without λ-transitions (empty-string transitions). Two nondeterministic finite automata A_1 and A_2 are *path equivalent*, also called multiset equivalent in [4], if for each string x, the number of distinct accepting computation paths (or accepting state transition sequences) for x by A_1 is equal to that for x by A_2. The path equivalence problem for nondeterministic finite automata (with λ-transitions) is *PSPACE*-hard. Using difference equations, however, Hunt and Stearns were able to show that the path equivalence problem for nondeterministic finite automata without λ-transitions is solvable in polynomial time [4]. We give an alternative

* Received by the editors November 22, 1989; accepted for publication (in revised form) May 28, 1991. Part of this paper appeared in "The equivalence and learning of probabilistic automata" presented at the 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, Research Triangle Park, North Carolina.

† Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York 11794. This research was supported in part by National Science Foundation grant CCR-8801575.

polynomial-time algorithm for this problem based on linear algebra. As a special case, the (language) equivalence problem for unambiguous finite automata is also polynomial-time solvable.

We also study the approximate equivalence (δ-equivalence) problem for probabilistic automata. Two probabilistic automata U_1 and U_2 are *δ-equivalent*, $\delta \geqq 0$, if for each string x the difference of its accepting probabilities by U_1 and U_2 is less than or equal to δ. A probabilistic automaton U is *positive* if for any two states q_1 and q_2 in U and any input symbol σ, the probability that U, on input σ, moves from state q_1 to state q_2 is greater than zero. We demonstrate an algorithm for the δ-equivalence problem for positive probabilistic automata. The algorithm will terminate except for the case where the input δ is equal to the maximum difference of accepting probabilities of a string.

2. Definitions. A (row) vector is *stochastic* if all its entries are greater than or equal to zero and sum to 1. A matrix is *stochastic* if all its row vectors are stochastic. Let $\mathcal{M}(i, j)$ be the set of all $(i \times j)$-dimensional stochastic matrices. Let λ be the empty string and $|x|$ be the length of string x. Let α^T be the transpose of the vector α. Let *span* be the function that maps a set of vectors to the vector space generated by the vectors in the set.

DEFINITION 2.1. A probabilistic automaton U is a 5-tuple (S, Σ, M, ρ, F), where $S = \{s_1, s_2, \cdots, s_n\}$ is a finite set of states, Σ is an input alphabet, M is a function from Σ into $\mathcal{M}(n, n)$, ρ is an n-dimensional stochastic row vector, and $F \subseteq S$ is a set of final states.

The vector ρ is called an *initial-state distribution* where the ith component of ρ indicates the probability of state s_i being the initial state. The value $M(\sigma)[i, j]$ is the probability that U moves from state s_i to state s_j after reading symbol $\sigma \in \Sigma$. We extend the domain of function M from Σ to Σ^* in the standard way, i.e., $M(x\sigma) = M(x)M(\sigma)$ for $x \in \Sigma^*$ and $\sigma \in \Sigma$, and $M(\lambda)$ is the $(n \times n)$-dimensional identity matrix. Let η_F be an n-dimensional row vector such that for $1 \leqq i \leqq n$,

$$\eta_F[i] = \begin{cases} 1 & \text{if } s_i \in F, \\ 0 & \text{otherwise.} \end{cases}$$

The *state distribution* induced by string x for U is

$$P_U(x) = \rho M(x),$$

where the ith component of $P_U(x)$ is the probability that U with initial-state distribution ρ moves to state s_i after reading x. We also define, for each string x,

$$Q_U(x) = M(x)(\eta_F)^T.$$

The *accepting probability* of x by U is

$$P_U(x)(\eta_F)^T = \rho Q_U(x),$$

which is the probability that U ends up in a final state when the input is string x.

DEFINITION 2.2. Let $U_1 = (S_1, \Sigma, M_1, \rho_1, F_1)$ and $U_2 = (S_2, \Sigma, M_2, \rho_2, F_2)$ be two probabilistic automata. Then U_1 and U_2 are said to be equivalent if for each string x, U_1 and U_2 accept x with equal probability, i.e., for all $x \in \Sigma^*$, $P_{U_1}(x)(\eta_{F_1})^T = P_{U_2}(x)(\eta_{F_2})^T$.

Let $U_1 = (S_1, \Sigma, M_1, \rho_1, F_1)$ and $U_2 = (S_2, \Sigma, M_2, \rho_2, F_2)$ be two probabilistic automata with number of states n_1 and n_2, respectively. We define, for each string x,

$$M_{U_1 \oplus U_2}(x) = \begin{bmatrix} M_1(x) & \mathbf{0}_{n_1 \times n_2} \\ \mathbf{0}_{n_2 \times n_1} & M_2(x) \end{bmatrix},$$

where $\mathbf{0}_{r\times s}$ is the $(r\times s)$-dimensional zero matrix; then we have, for any $\sigma\in\Sigma$,

$$M_{U_1\oplus U_2}(x\sigma)=M_{U_1\oplus U_2}(x)M_{U_1\oplus U_2}(\sigma).$$

We also define, for each string x,

$$\mathbf{P}_{U_1\oplus U_2}(x)=[\rho_1,\rho_2]M_{U_1\oplus U_2}(x).$$

Since the issue of computing with real numbers is subtle, in the rest of this paper we assume that all inputs consist of rational numbers and that each arithmetic operation on rational numbers can be done in constant time, unless stated otherwise.

3. Equivalence of probabilistic automata. In this section we present a polynomial-time algorithm for the equivalence problem for probabilistic automata.

THEOREM 3.1. *There is an algorithm running in time $O((n_1+n_2)^4)$ that takes as input two probabilistic automata U_1 and U_2 and determines whether U_1 and U_2 are equivalent, where n_1 and n_2 are the number of states in U_1 and U_2, respectively. Furthermore, if U_1 and U_2 are not equivalent then the algorithm outputs the lexicographically minimum string which is accepted by U_1 and U_2 with different probabilities. This string will always be of length at most n_1+n_2-1.*

It has been shown [7] that two probabilistic automata are not equivalent if and only if there exists a string x of length at most n_1+n_2-1 such that $P_{U_1}(x)(\eta_{F_1})^T\neq P_{U_2}(x)(\eta_{F_2})^T$. This implies that the complexity of the equivalence problem for probabilistic automata is in *coNP*.

For two probabilistic automata U_1 and U_2, let

$$H(U_1,U_2)=\{P_{U_1\oplus U_2}(x): x\in\Sigma^*\}.$$

Recall that U_1 and U_2 are equivalent if and only if

$$\forall x\in\Sigma^*,\quad P_{U_1}(x)(\eta_{F_1})^T=P_{U_2}(x)(\eta_{F_2})^T.$$

We can reformulate this equation as

$$\forall x\in\Sigma^*,\quad P_{U_1\oplus U_2}(x)[\eta_{F_1},-\eta_{F_2}]^T=0.$$

LEMMA 3.2. *Let U_1 and U_2 be two probabilistic automata. If V is a basis for span $(H(U_1,U_2))$ then U_1 and U_2 are equivalent if and only if for all $\mathbf{v}\in V$, $\mathbf{v}[\eta_{F_1},-\eta_{F_2}]^T=0$.*

Proof. Since the proof is straightforward, we omit it here. □

Because the dimension of the vector space $span(H(U_1,U_2))$ is at most n_1+n_2, the number of elements in V is at most n_1+n_2. A basic idea behind the design of our polynomial-time algorithm for the equivalence problem for probabilistic automata is to find a basis $V\subseteq H(U_1,U_2)$ for the vector space $span(H(U_1,U_2))$. If we are able to find such a basis in polynomial time then we can solve the equivalence problem for probabilistic automata in polynomial time.

Proof of Theorem 3.1. Without loss of generality, we let $\Sigma=\{0,1\}$. We define a binary tree T as follows. Tree T will have a node for every string in Σ^*. The root of T is $node(\lambda)$. Every $node(x)$ (where x is a string) in T has two children $node(x0)$ and $node(x1)$. Let $P_{U_1\oplus U_2}(x)$ be the (n_1+n_2)-dimensional vector associated with $node(x)$. For $node(x\sigma)$, $\sigma\in\Sigma$, its associated vector $P_{U_1\oplus U_2}(x\sigma)$ can be calculated by multiplying its parent's associated vector $P_{U_1\oplus U_2}(x)$ by $M_{U_1\oplus U_2}(\sigma)$.

The method we use to determine whether U_1 and U_2 are equivalent is to prune tree T. Initially, we set V to be the empty set. We then visit the nodes in T in *breadth-first* order. At each node $node(x)$, we verify whether its associated vector $P_{U_1\oplus U_2}(x)$ is linearly independent of V. If it is, we add the vector to V. Otherwise, we prune the subtree rooted at $node(x)$. We stop traversing tree T when every node in T is either visited or pruned. The vectors in the resulting set V will be linearly independent. We

will show in Lemma 3.3 that the vectors in V form a basis for $span(H(U_1, U_2))$. The traversal order does not affect whether V is a basis for $span(H(U_1, U_2))$. A different traversal order will simply generate a different basis set. When U_1 and U_2 are not equivalent, a breadth-first traversal is necessary for finding the lexicographically minimum string whose accepting probabilities by U_1 and U_2 are different.

Our tree pruning algorithm appears in Table 1. In the algorithm, *queue* is a queue and N is a set of nodes. At the end of the algorithm, we verify whether $span(V)$ is a null space of linear transformation $[\eta_{F_1}, -\eta_{F_2}]^T$. If it is, then U_1 and U_2 are equivalent. Otherwise, we return the lexicographically minimum string in the set $\{x: node(x) \in N\}$ which is accepted by U_1 and U_2 with different probabilities.

Correctness. Let T_N be the tree formed by the nodes in

$$N \cup \{node(x\sigma): node(x) \in N, \sigma \in \Sigma\}$$

(the set of nodes that have been visited). Because the vectors in V are (n_1+n_2)-dimensional, T_N has at most n_1+n_2 internal nodes (those in N) and at most n_1+n_2+1 leaves. Set V consists of the vectors associated with the internal nodes of T_N. Since we prune tree T at $node(x)$ when $P_{U_1 \oplus U_2}(x) \in span(V)$, the vectors associated with the leaves of T_N will be linearly dependent to the vectors in V. For example, in Fig. 1, the associated vectors $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_6$ of the internal nodes are linearly independent and the associated vectors of the leaves marked by $\otimes$ are linearly dependent to $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_6$.

TABLE 1
Algorithm for equivalence of probabilistic automata.

```
Input: U1 = (S1, {0, 1}, M1, ρ1, F1), U2 = (S2, {0, 1}, M2, ρ2, F2);
1.     Set V and N to be the empty set;
2.     queue ← node(λ);
3.     while queue is not empty do
4.     begin take an element node(x) from queue;
5.           if P_{U1⊕U2}(x) ∉ span(V) then
6.           begin add node(x0) and node(x1) to queue;
7.                 add vector P_{U1⊕U2}(x) to V;
8.                 add node(x) to N
               end;
       end;
9.     if ∀v ∈ V, v[η_{F1}, −η_{F2}]^T = 0 then return(yes)
10.    else return (lex-min {x: node(x) ∈ N, P_{U1⊕U2}(x)[η_{F1}, −η_{F2}]^T ≠ 0});
```

We will prove that the vectors in the resulting set V form a basis for the vector space $span(H(U_1, U_2))$. For $i \geqq 0$, let

$$V_i = \{P_{U_1 \oplus U_2}(xy): node(x) \text{ is a leaf}, |y| = i\}.$$

Set V_0 is the set of vectors associated with the leaves of T_N and set V_i, $i \geqq 1$, is the set of vectors associated with the unvisited nodes of T which are of distance i from a leaf. It can be seen that

$$span\left(V \cup \bigcup_{i=0}^{\infty} V_i\right) = span(\{P_{U_1 \oplus U_1}(x): x \in \Sigma^*\}) = span(H(U_1, U_2)).$$

LEMMA 3.3. *For all $i \geqq 0$, $V_i \subseteq span(V)$.*

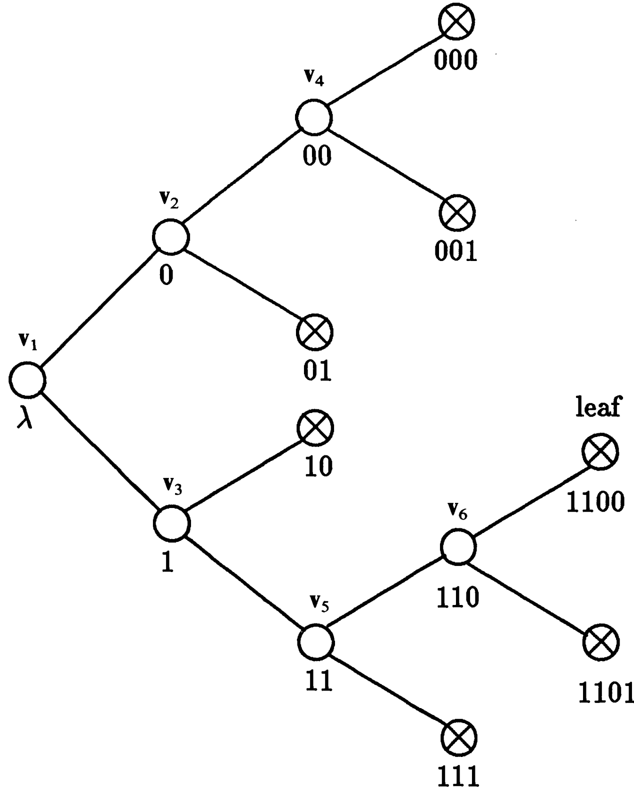


FIG. 1. *A T_N tree.*

Proof. Let $V=\{\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_r\}$ for some $r \leqq n_1+n_2$. We prove this lemma by induction on i. The base case $V_0 \subseteq \{\mathbf{v}M_{U_1 \oplus U_2}(\sigma): \mathbf{v} \in V, \sigma \in \Sigma\} \subseteq span(V)$ follows from the algorithm. Assume that $V_i \subseteq span(V)$. Then for any x, y, and $\sigma \in \Sigma$ such that $node(x)$ is a leaf and $|y|=i$ it will be the case that

$$P_{U_1 \oplus U_2}(xy\sigma) = P_{U_1 \oplus U_2}(xy) M_{U_1 \oplus U_2}(\sigma) = \left(\sum_{i=1}^{r} m_i \mathbf{v}_i \right) M_{U_1 \oplus U_2}(\sigma)$$

$$= \sum_{i=1}^{r} m_i (\mathbf{v}_i M_{U_1 \oplus U_2}(\sigma)) \in span(V \cup V_0) = span(V). \qquad \square$$

Thus the vectors in V form a basis for $span(H(U_1, U_2))$. By Lemma 3.2, the algorithm can determine whether U_1 and U_2 are equivalent by testing whether for all $\mathbf{v} \in V$, $\mathbf{v}[\eta_{F_1}, -\eta_{F_2}]^T = 0$ holds.

LEMMA 3.4. *Let U_1 and U_2 be two probabilistic automata having number of states n_1 and n_2, respectively. If U_1 and U_2 are not equivalent then the algorithm in Table 1 outputs the lexicographically minimum string which is accepted by U_1 and U_2 with different probabilities. This string will be of length at most n_1+n_2-1.*

Proof. Let *lex* be the function that maps a string to its lexicographic order in Σ^*. Let x be the string returned by our algorithm. Assume that the lemma is false. Let y be the lexicographically minimum string such that $lex(y) < lex(x)$ and such that $P_{U_1 \oplus U_2}(y)[\eta_{F_1}, -\eta_{F_2}]^T \neq 0$. Since $node(y) \notin N$, there must exist a leaf $node(z)$ such that $y = zw$ for some $w \in \Sigma^*$. Since we used a breadth-first traversal in our tree pruning

algorithm, the associated vector $P_{U_1\oplus U_2}(z)$ of $node(z)$ will be in $span(\{P_{U_1\oplus U_2}(u)\colon u\in \Sigma^*, lex(u)<lex(z)\})$. Hence it will be the case that

$$
\begin{aligned}
P_{U_1\oplus U_2}(y)[\eta_{F_1}, -\eta_{F_2}]^T &= P_{U_1\oplus U_2}(z)M_{U_1\oplus U_2}(w)[\eta_{F_1}, -\eta_{F_2}]^T \\
&= \sum_{lex(u)<lex(z)} m_u P_{U_1\oplus U_2}(u)M_{U_1\oplus U_2}(w)[\eta_{F_1}, -\eta_{F_2}]^T \qquad (1)\\
&= \sum_{lex(u)<lex(z)} m_u P_{U_1\oplus U_2}(uw)[\eta_{F_1}, -\eta_{F_2}]^T.
\end{aligned}
$$

Because $lex(uw)<lex(zw)=lex(y)$ for any $lex(u)<lex(z)$, the value of equation (1) is zero. This contradicts the assumption $P_{U_1\oplus U_2}(y)[\eta_{F_1}, -\eta_{F_2}]^T\neq 0$. Therefore the string x returned by our algorithm will be the lexicographically minimum string whose accepting probabilities by U_1 and U_2 are different. Furthermore, since no node in N is labeled by a string of length $>n_1+n_2-1$, the length of string x will be at most n_1+n_2-1. □

Complexity. Recall that we assume an arithmetic operation on rational numbers can be done in constant time. The binary tree T_N has at most n_1+n_2 internal nodes and thus at most n_1+n_2+1 leaves. The vector associated with $node(x\sigma)$ is calculated by multiplying its parent's associated vector $P_{U_1\oplus U_2}(x)$ by $M_{U_1\oplus U_2}(\sigma)$, which can be done in time $O((n_1+n_2)^2)$. To verify whether a set of (n_1+n_2)-dimensional vectors is linearly independent needs time $O((n_1+n_2)^3)$ [1]. Thus the total runtime is $O((n_1+n_2)^4)$. This completes the proof of Theorem 3.1. □

4. Equivalence of uninitiated probabilistic automata. In this section we show that the covering and equivalence problems for uninitiated probabilistic automata are also polynomial-time solvable.

DEFINITION 4.1. An uninitiated probabilistic automaton U is a 4-tuple (S, Σ, M, F), where S is a finite set of n states, Σ is an input alphabet, M is a function from Σ into $\mathcal{M}(n, n)$, and $F\subseteq S$ is a set of final states.

Given an initial-state distribution ρ, we can form a probabilistic automaton $U(\rho)$ out of uninitiated automaton U.

DEFINITION 4.2. Let U_1 and U_2 be two uninitiated probabilistic automata. Then U_1 is said to cover U_2 if for any initial-state distribution ρ_2 for U_2 there is an initial-state distribution ρ_1 for U_1 such that $U_1(\rho_1)$ and $U_2(\rho_2)$ are equivalent.

DEFINITION 4.3. Let U_1 and U_2 be two uninitiated probabilistic automata. Then U_1 and U_2 are said to be equivalent if U_1 covers U_2 and U_2 covers U_1.

Let $U_1=(S_1, \Sigma, M_1, F_1)$ and $U_2=(S_2, \Sigma, M_2, F_2)$ be two uninitiated probabilistic automata having n_1 and n_2 states, respectively. Recall that $Q_{U_1}(x)=M_1(x)(\eta_{F_1})^T$. Let J be an $(n_1\times r)$-dimensional matrix whose column vectors $Q_{U_1}(x_1)$, $Q_{U_1}(x_2), \cdots, Q_{U_1}(x_r)$ for some $r\leqq n_1$ form a basis for the vector space $span(\{Q_{U_1}(x)\colon x\in \Sigma^*\})$. Let G be the $(n_2\times r)$-dimensional matrix whose column vectors are $Q_{U_2}(x_1)$, $Q_{U_2}(x_2), \cdots, Q_{U_2}(x_r)$. It has been shown [7] that if B is an $(n_2\times n_1)$-dimensional stochastic matrix such that $BJ=G$ then U_1 covers U_2 if and only if the condition for all $\sigma\in\Sigma$, $BM_1(\sigma)J=M_2(\sigma)G$ holds. Stochastic matrix B is a transformation from initial-state distributions for U_2 to those for U_1. The condition that for all $\sigma\in\Sigma$, $BM_1(\sigma)J=M_2(\sigma)G$ guarantees that $U_2(\rho_2)$ and $U_1(\rho_2 B)$ are equivalent for any initial-state distribution ρ_2 for U_2. Since ρ_2 and B are stochastic, so is $\rho_2 B$.

LEMMA 4.4 [7]. *Let U_1 and U_2 be two uninitiated probabilistic automata and let J and G be defined as above. Then U_1 covers U_2 if and only if there exists an $(n_2\times n_1)$-dimensional stochastic matrix B satisfying $BJ=G$ and if for any such B the condition $BM_1(\sigma)J=M_2(\sigma)G$ holds for all $\sigma\in\Sigma$.*

The problem of finding a stochastic matrix B such that $BJ = G$ can be reduced to the linear programming problem because of the "stochastic" restriction on B. If no such B exists then U_1 does not cover U_2. Once B has been found, it is easy to verify the condition. Thus our result implies the following.

THEOREM 4.5. *There is a polynomial-time algorithm that takes as input two uninitiated probabilistic automata U_1 and U_2 and determines whether U_1 covers U_2.*

Proof. By Lemma 4.4, we need to find matrices J, G, and B and then verify the condition for all $\sigma \in \Sigma$, $BM_1(\sigma)J = M_2(\sigma)G$. Using the idea that was used in the algorithm in Table 1, we can find in polynomial time strings $x_1, x_2, \cdots, x_r$ for some $r \leqq n_1$ such that $Q_{U_1}(x_1), Q_{U_1}(x_2), \cdots, Q_{U_1}(x_r)$ form a basis for $span(\{Q_{U_1}(x): x \in \Sigma^*\})$. Then matrices J and G can be calculated easily. The problem of finding a stochastic matrix B such that $BJ = G$ can be reduced to the linear programming problem, which is well known to be solvable in polynomial time [5]-[6]. If B does not exist, then U_1 does not cover U_2. Otherwise, we need to verify whether the condition for all $\sigma \in \Sigma$, $BM_1(\sigma)J = M_2(\sigma)G$ holds. If it does, then U_1 covers U_2. Otherwise, U_1 does not cover U_2. □

THEOREM 4.6. *There is a polynomial-time algorithm that takes as input two uninitiated probabilistic automata U_1 and U_2 and determines whether U_1 and U_2 are equivalent.*

Proof. By the definition of equivalence for uninitiated probabilistic automata, we need to verify that U_1 covers U_2 and that U_2 covers U_1. Theorem 4.5 implies that these tasks can be done in polynomial time. □

5. Approximate equivalence. In this section we consider the problem of determining whether two probabilistic automata are δ-equivalent. For $\delta \geqq 0$, two probabilistic automata are δ-equivalent if for every string x the probabilities that the two automata accept x differ by at most δ. Two equivalent probabilistic automata are zero-equivalent and any two probabilistic automata are 1-equivalent.

DEFINITION 5.1. Let $U_1 = (S_1, \Sigma, M_1, \rho_1, F_1)$ and $U_2 = (S_2, \Sigma, M_2, \rho_2, F_2)$ be two probabilistic automata and $\delta \geqq 0$ be a real number. Then U_1 and U_2 are said to be δ-equivalent if for all $x \in \Sigma^*$, $|P_{U_1}(x)(\eta_{F_1})^T - P_{U_2}(x)(\eta_{F_2})^T| \leqq \delta$.

We do not know the precise complexity for the δ-equivalence problem in general. In the following we consider the δ-equivalence problem for positive probabilistic automata only. A matrix is *positive* if all its entries are greater than zero. A probabilistic automaton $U = (S, \Sigma, M, \rho, F)$ is *positive* if all its transition matrices $M(\sigma)$, $\sigma \in \Sigma$, are positive. The reason for the restriction to positive matrices is that under this restriction the accepting probabilities of long strings can be estimated using the accepting probabilities of short strings [8].

In order to explain how we can estimate the accepting probabilities of long strings using short strings we will need an additional definition and lemma. For two probabilistic automata $U_1 = (S_1, \Sigma, M_1, \rho_1, F_1)$ and $U_2 = (S_2, \Sigma, M_2, \rho_2, F_2)$, we define

$$\delta^*(U_1, U_2) = \sup \{|P_{U_1}(\eta_{F_1})^T - P_{U_2}(\eta_{F_2})^T|: x \in \Sigma^*\}$$

and

$$\Delta(U_1, U_2) = \min_{i,j,k,\sigma} \{M_i(\sigma)[j, k]\}.$$

For a row vector $\alpha = [\alpha_1, \sigma_2, \cdots, \alpha_n]$, we define

$$|\sigma| = \max_{1 \leqq i \leqq n} \{\alpha_i\} \text{ and } \|\alpha\| = \max_{1 \leqq j,k \leqq n} \{|\alpha_j - \alpha_k|\}.$$

LEMMA 5.2 [8]. *Let α be a row vector, let A be a stochastic matrix, and let τ be $\min_{i,j} \{A[i,j]\}$. Then $|A\alpha^T - \alpha^T| \leqq \|\alpha\|$ and $\|A\alpha^T\| \leqq (1-2\tau)\|\alpha\|$.*

Let U_1 and U_2 be two positive probabilistic automata. We will show how to estimate the value $\delta^*(U_1, U_2)$. For any $0 < \varepsilon \leqq 1$, let $m = O(\log(2/\varepsilon)/\Delta(U_1, U_2))$ such that $(1-2\Delta(U_1, U_2))^m \leqq \varepsilon/2$. For any string $x = yz$ such that $|x| \geqq m$ and $|z| = m$ it will be the case that

$$\begin{aligned}
&|\rho_1 Q_{U_1}(x) - \rho_2 Q_{U_2}(x)| \\
&\quad \leqq |\rho_1 Q_{U_1}(x) - \rho_1 Q_{U_1}(z)| + |\rho_2 Q_{U_2}(x) - \rho_2 Q_{U_2}(z)| \\
&\qquad + |\rho_1 Q_{U_1}(z) - \rho_2 Q_{U_2}(z)| \\
&\quad \leqq |Q_{U_1}(x) - Q_{U_1}(z)| + |Q_{U_2}(x) - Q_{U_2}(z)| + \delta' \\
&\quad = |M_1(y) Q_{U_1}(z) - Q_{U_1}(z)| + |M_2(y) Q_{U_2}(z) - Q_{U_2}(z)| + \delta' \\
&\quad \leqq \|Q_{U_1}(z)\| + \|Q_{U_2}(z)\| + \delta' \\
&\quad \leqq (1-2\Delta(U_1, U_2))^m \|\eta_{F_1}\| + (1-2\Delta(U_1, U_2))^m \|\eta_{F_2}\| + \delta' \\
&\quad \leqq \varepsilon/2 + \varepsilon/2 + \delta' \\
&\quad = \varepsilon + \delta'
\end{aligned}$$

where $\delta' = \max\{|\rho_1 Q_{U_1}(w) - \rho_2 Q_{U_2}(w)|: |w| \leqq m\}$. Since $\delta' \leqq \delta^*(U_1, U_2)$, we have $|\delta' - \delta^*(U_1, U_2)| \leqq \varepsilon$. Thus for any arbitrarily small value $\varepsilon > 0$ there is an m large enough such that the difference between $\delta' = \max\{|\rho_1 Q_{U_1}(x) - \rho_2 Q_{U_2}(x)|: |x| \leqq m\}$ and $\delta^*(U_1, U_2)$ is $\leqq \varepsilon$.

We consider the following two questions.

Question 1. Given two positive probabilistic automata U_1 and U_2 and a number $0 < \varepsilon \leqq 1$, find a number δ' such that $|\delta' - \delta^*(U_1, U_2)| \leqslant \varepsilon$.

Question 2. Given two positive probabilistic automata U_1 and U_2 and a number $0 \leqq \delta \leqq 1$, determine whether U_1 and U_2 are δ-equivalent.

Our result for the first question is that for any two positive probabilistic automata U_1 and U_2 and for any $\varepsilon > 0$, we can find in exponential time a δ' such that $|\delta' - \delta^*(U_1, U_2)| < \varepsilon$.

THEOREM 5.3. *There exists an algorithm that takes as input two positive probabilistic automata U_1 and U_2 together with a number $\varepsilon (0 < \varepsilon \leqq 1)$ and outputs a number δ' such that $|\delta' - \delta^*(U_1, U_2)| \leqq \varepsilon$. In addition, the algorithm runs in time $O((2/\varepsilon)^{O(\log(k)/\Delta)})$ where Δ is the minimum entry of the transition matrices for U_1 and U_2 and k is the size of the input alphabet of U_1 and U_2.*

Proof. Let $(1-2\Delta)^m \leqq \varepsilon/2$. We evaluate $|\rho_1 Q_{U_1}(x) - \rho_2 Q_{U_2}(x)|$ for all strings x of length $\leqq m$ and let $\delta' = \max\{|\rho_1 Q_{U_1}(x) - \rho_2 Q_{U_2}(x)|: |x| \leqq m\}$. By the above discussion, we can see that $|\delta' - \delta^*(U_1, U_2)| \leqq \varepsilon$.

Since $m = O(\log(2/\varepsilon)/\Delta)$, we have to evaluate $k^m - 1 = (2/\varepsilon)^{O(\log(k)/\Delta)}$ values. For each string x such that $|x| \leqq m$, $\rho_1 Q_{U_1}(x)$ and $\rho_2 Q_{U_2}(x)$ can be evaluated in time $O(m \cdot n_1^2 \cdot \log(2/\Delta) + m \cdot n_2^2 \cdot \log(2/\Delta))$, where n_1 and n_2 are the number of states in U_1 and U_2, respectively. Therefore the total runtime is

$$O((2/\varepsilon)^{O(\log(k)/\Delta)} \cdot m \cdot (n_1^2 + n_2^2) \cdot \log(2/\Delta)) = O((2/\varepsilon)^{O(\log(k)/\Delta)} \cdot (n_1^2 + n_2^2)).$$

Since we can assume that $n_1, n_2 \leqq 1/\Delta$, the runtime can be simplified to

$$O((2/\varepsilon)^{O(\log(k)/\Delta)}). \qquad \square$$

For the δ-equivalence problem for positive probabilistic automata (the second question), we first consider the case where $\delta \neq \delta^*(U_1, U_2)$. We let

$$N = \min\{n: |\delta - \delta^*(U_1, U_2)| \geqq 1/n, \ n \text{ is a natural number}\}$$

and $\delta' = \max\{|\rho_1 Q_{U_1}(x) - \rho_2 Q_{U_2}(x)|: |x| \leqq m\}$, where $(1-2\Delta(U_1, U_2))^m < 1/(2N)$. If $\delta' < \delta$, then

$$\delta^*(U_1, U_2) \leqq \delta' + 2(1-2\Delta(U_1, U_2))^m < \delta' + 1/N \leqq \delta.$$

If $\delta' > \delta$, then $\delta^*(U_1, U_2) > \delta$.

It seems difficult, however, to deal with the case where $\delta = \delta^*(U_1, U_2)$, except for the cases where $\delta = 1$ and $\delta = 0$ which can be solved by our previous algorithm. The main obstacle for the case where $\delta = \delta^*(U_1, U_2)$ is that the value $\delta^*(U_1, U_2)$ may occur in the limit in such a way that it may not be possible to calculate it in finite steps. Consider, for instance, the following example from [8]. Let $U' = (\{s_1, s_2\}, \{0, 1\}, M', [1, 0], \{s_2\})$, where

$$M'(0) = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \quad \text{and} \quad M'(1) = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ 0 & 1 \end{bmatrix}.$$

It is easy to verify that for string $x = \sigma_1\sigma_2 \cdots \sigma_n$, $\sigma_i \in \Sigma$, $1 \leqq i \leqq n$, the accepting probability of x by U is

$$[1, 0]M'(x)[0, 1]^T = \frac{\sigma_n}{2} + \frac{\sigma_{n-1}}{2^2} + \cdots + \frac{\sigma_1}{2^n}.$$

Let U'' be the probabilistic automaton that accepts every string with probability $\frac{1}{3}$. Then we have $\delta^*(U', U'') = \frac{2}{3}$, which occurs when $x = 111 \cdots$.

Our result about the δ-equivalence problem for positive probabilistic automata is as follows.

THEOREM 5.4. *There exists an algorithm which takes as input two positive probabilistic automata U_1 and U_2 and a number $\delta (0 \leqq \delta \leqq 1)$ and determines whether U_1 and U_2 are δ-equivalent when $\delta \neq \delta^*(U_1, U_2)$. In addition, when the algorithm terminates, it runs in time $O((\max\{2, N\})^{O(\log(k)/\Delta)})$ where N is the minimum integer such that $|\delta - \delta^*(U_1, U_2)| \geqq 1/N$, Δ is the minimum entry of transition matrices for U_1 and U_2 and k is the size of the input alphabet of U_1 and U_2.*

Proof. Our algorithm appears in Table 2. When $\delta = \delta^*(U_1, U_2)$, the conditions in steps 6 and 7 will not hold and the algorithm will not terminate. The complexity analysis is similar to that of Theorem 5.3. □

6. Path equivalence of nondeterministic finite automata without λ-transitions. In this section we apply our algorithm to the path equivalence problem for nondeterministic finite automata without λ-transitions. We first give definitions of nondeterministic finite automata, computation paths, and path equivalence.

TABLE 2
Algorithm for δ-equivalence of probabilistic automata.

```
Input: U1 = (S1, Σ, M1, ρ1, F1), U2 = (S2, Σ, M2, ρ2, F2), 0 ≦ δ ≦ 1;
1.  Δ ← min_{σ,i,j,k} {M_i(σ)[j, k]};
2.  if δ = 1 then return (yes);
3.  if δ = 0 then run the algorithm in Table 1;
4.  i ← 0;
5.  while true do
6.  begin if ∃x, |x| = i, |ρ1 Q_U1(x) − ρ2 Q_U2(x)| > δ then return (no);
7.        if max {|ρ1 Q_U1(x) − ρ2 Q_U2(x)|: |x| = i} + 2(1 − 2Δ)^i ≦ δ
          then return (yes);
8.        i ← i + 1;
9.  end;
```

DEFINITION 6.1. A nondeterministic finite automaton A is a 5-tuple $(S, \Sigma, \delta, s_1, F)$, where S is a finite set of states, Σ is an input alphabet, δ is a transition function from $S \times (\Sigma \cup \{\lambda\})$ into the power set of S, s_1 is the initial state, and $F \subseteq S$ is a set of final states.

DEFINITION 6.2. Let $A = (S, \Sigma, \delta, s_1, F)$ be a nondeterministic finite automaton. A computation path θ for A is a finite nonempty sequence

$$((q_1, a_1), (q_2, a_2), \cdots, (q_n, a_n), q_{n+1}),$$

where $(q_i, a_i) \in S \times (\Sigma \cup \{\lambda\})$ for $1 \leqq i \leqq n$, $q_{n+1} \in S$, and $q_{i+1} \in \delta(q_i, a_i)$ for $1 \leqq i \leqq n$. If $q_1 = s_1$ and $q_{n+1} \in F$ then the sequence θ is said to be an accepting computation path for x, where $x = a_1 a_2 \cdots a_n$.

A nondeterministic finite automaton with λ-transitions could accept a finite string via an infinite number of computation paths. Consequently, the path equivalence (called multiset equivalence in [4]) of nondeterministic finite automata is defined as follows.

DEFINITION 6.3. Let A_1 and A_2 be two nondeterministic finite automata. Then A_1 and A_2 are said to be path equivalent if for each string x the number of distinct accepting computation paths for x by A_1 is equal to that for x by A_2 or both are infinite.

The path equivalence problem for nondeterministic finite automata (with λ-transitions) is *PSPACE*-hard. We do not know whether the problem is decidable.

THEOREM 6.4. *The following problem is PSPACE-hard: Given two nondeterministic finite automata (with λ-transitions) A_1 and A_2, determine whether A_1 and A_2 are path equivalent.*

Proof. We reduce the *PSPACE*-complete problem of determining whether a nondeterministic finite automaton accepts all strings [3] to this problem. For each nondeterministic finite automaton $A = (S, \Sigma, \delta, s_1, F)$, we construct the following two automata A_1 and A_2. Automaton A_1 is $(S, \Sigma, \delta', s_1, F)$, where $\delta'(s_1, \lambda) = s_1$ and $\delta'(q, a) = \delta(q, a)$ for any $q \in S$ and $a \in \Sigma \cup \{\lambda\}$. A string is accepted by A if and only if it is accepted by A_1 via an infinite number of computation paths. Automaton A_2 is $(\{s_1\}, \Sigma, \delta_2, s_1, \{s_1\})$, where $\delta_2(s_1, a) = s_1$ for any $d \in \Sigma \cup \{\lambda\}$, which accepts every string via an infinite number of computation paths. It is easy to see that A accepts all strings if and only if A_1 and A_2 are path equivalent. □

Therefore we only consider the path equivalence problem for nondeterministic finite automata without λ-transitions. Let $A = (S, \Sigma, \delta, s_1, F)$ be an n-state nondeterministic finite automaton without λ-transitions. The transition function δ can be represented as $(n \times n)$-dimensional transition matrices $M(\sigma)$, $\sigma \in \Sigma$, such that

$$M(\sigma)[i, j] = \begin{cases} 1 & \text{if } s_j \in \delta(s_i, \sigma), \\ 0 & \text{otherwise.} \end{cases}$$

Note that the above transition matrices $M(\sigma)$ are not stochastic. For each string x, $P_A(x)\eta_F$, which is $[1, 0, 0, \cdots, 0]M(x)\eta_F$, is equal to the number of distinct accepting computation paths for x by A.

The following result about the path equivalence problem for nondeterministic finite automata without λ-transitions was first proved in [4] (cf. [9]); we give a different proof using our techniques.

THEOREM 6.5. *There is a polynomial-time algorithm that takes as input two nondeterministic finite automata A_1 and A_2 without λ-transitions and determines whether A_1 and A_2 are path equivalent. If A_1 and A_2 are not path equivalent then the algorithm outputs the lexicographically minimum string x which is accepted by A_1 and A_2 via different*

numbers of computation paths. The length of x will be less than the total number of states in A_1 and A_2.

Proof. We use transition matrices to represent the transition functions of nondeterministic finite automata as described earlier and apply the algorithm in Table 1 to verify whether A_1 and A_2 are path equivalent. The rest of the proof is identical to that of Theorem 3.1. □

7. Equivalence of unambiguous finite automata. A nondeterministic finite automaton A is of *finite degree of ambiguity* if there exists a constant k such that every string is accepted by A via at most k distinct computation paths. If k is 1 then the automaton is *unambiguous.*

In [9] Stearns and Hunt showed that, for any fixed k, there exists a polynomial-time algorithm for the (language) equivalence problem for nondeterministic finite automata of degree of ambiguity less than or equal to k. Our algorithm is not applicable to the problem in general. It can, however, solve the (language) equivalence problem for unambiguous finite automata in polynomial time, because two unambiguous finite automata are (language) equivalent if and only if they are path equivalent.

The result of Theorem 7.2 about the equivalence problem for unambiguous finite automata was first proved in [9]; we give a different proof using our techniques.

LEMMA 7.1 [9]. *There is a polynomial-time algorithm that takes as input an n-state nondeterministic finite automaton A with λ-transitions and outputs an equivalent nondeterministic finite automaton A' without λ-transitions and having at most n states. In addition, if A is of degree of ambiguity k then A' is of degree of ambiguity at most k.*

THEOREM 7.2. *There is a polynomial-time algorithm that takes as input two unambiguous finite automata A_1 and A_2 and determines whether A_1 and A_2 are equivalent. If A_1 and A_2 are not equivalent then the algorithm outputs the lexicographically minimum string which is accepted by A_1, but not by A_2, or vice versa. Furthermore, this string will be of length less than the total number of states in A_1 and A_2.*

Proof. By Lemma 7.1, we can transform A_1 and A_2 to their equivalent unambiguous finite automata A_1' and A_2' without λ-transitions, respectively. Although the transformation does not preserve the number of accepting computation paths for each string in the case of general nondeterministic finite automata, it does preserve the number of accepting computation paths for each string in the case of unambiguous finite automata. We then apply the algorithm in Table 1 to verify whether A_1' and A_2' are path equivalent. The rest of the proof is identical to that of Theorem 3.1. □

8. Conclusion. In the complexity analysis of our algorithm for the equivalence problem for probabilistic automata, we assumed that each arithmetic operation on rational numbers could be done in constant time. This assumption, however, is not essential to the polynomial execution time of our algorithm. It is possible to show that the number of bits in the vectors associated with the nodes in tree T_N grows polynomially and that our algorithm will still run in polynomial time if arithmetic operations on rational numbers require time proportional to their number of bits.

A probabilistic automaton U with a real-valued cut-point π defines the language consisting of those strings accepted by U with probability greater than π. The equivalence problem for cut-point probabilistic automata is quite different from the (exact) equivalence problem for probabilistic automata because the emptiness problem for cut-point probabilistic automata is undecidable [7]. This undecidability result holds even if all input numbers are rational [2]. By a simple reduction, we can see that the equivalence problem for cut-point probabilistic automata is *undecidable* even if all input numbers are rational.

Acknowledgments. The author would like to thank Professor Ker-I Ko for his support and for many helpful discussions, Professor Scott Smolka, who introduced him to the problem of determining the equivalence of probabilistic automata, and the referees for their valuable comments.

REFERENCES

[1] D. K. FADDEEV AND V. N. FADDEEVA, *Computational Methods of Linear Algebra*, Freeman, San Francisco, 1963.
[2] S. GINSBURG, *The Mathematical Theory of Context-Free Languages*, McGraw-Hill, New York, 1966.
[3] J. HOPCROFT AND J. ULLMAN, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.
[4] H. B. HUNT III AND R. E. STEARNS, *On the complexity of equivalence, nonlinear algebra, and optimization on rings, semirings, and lattices*, extended abstract, Computer Science Department, State University of New York, Albany, NY, TR 86-23, 1986.
[5] N. KARMARKAR, *A new polynomial-time algorithm for linear programming*, Combinatorica, 4 (1984), pp. 373-395.
[6] L. G. KHACHIYAN, *A polynomial algorithm in linear programming*, Soviet Math. Dokl., 20 (1979), pp. 191-194.
[7] A. PAZ, *Introduction to Probabilistic Automata*, Academic Press, New York, 1971.
[8] M. O. RABIN, *Probabilistic automata*, Inform. Control, 6 (1963), pp. 230-245.
[9] R. E. STEARNS AND H. B. HUNT III, *On the equivalence and containment problems for unambiguous regular expressions, regular grammars, and finite automata*, SIAM J. Comput., 14 (1985), pp. 598-611.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 228-239, April 1992
© 1992 Society for Industrial and Applied Mathematics
003

SUBGROUP REFINEMENT ALGORITHMS FOR ROOT FINDING IN $GF(q)$*

A. J. MENEZES†, P. C. VAN OORSCHOT‡, AND S. A. VANSTONE‡

Abstract. This paper presents a generalization of Moenck's root finding algorithm over $GF(q)$, for q a prime or prime power. The generalized algorithm, like its predecessor, is deterministic, given a primitive element ω for $GF(q)$. If $q-1$ is b-smooth, where $b=(\log q)^{O(1)}$, then the algorithm runs in polynomial time. An analogue of this generalization which applies to extension fields $GF(q^m)$ is also considered. The analogue is a deterministic algorithm based on the recently introduced affine method for root finding in $GF(q^m)$, where $m>1$; it is, however, less efficient that the affine method itself.

Key words. root finding, polynomial factorization, finite fields

AMS(MOS) subject classifications. primary 68Q40; secondary 11Y16, 11J06, 05B25

1. Introduction. This paper is concerned with finding roots of polynomials in finite fields. That is, given a polynomial $f(x)$ over $GF(q)$, find all elements $\alpha \in GF(q)$ such that $f(\alpha)=0$. If the degree of $f(x)$ is n, we will assume, without loss of generality, that $f(x)$ has n distinct roots in $GF(q)$. The root finding problem has received considerable attention and a number of efficient algorithms have been devised, (e.g., [2]-[6], [10], [14], [17]).

An algorithm proposed by Moenck applies to fields $GF(q)$ when q has the special form $q-1=T\cdot 2^t$ where T is about the size of t. Moenck shows that in this special case, given a primitive element for the field, the roots of a polynomial can be found in polynomial time; he provides an efficient algorithm for doing so. We generalize this idea to q having the form $q-1=T\cdot p^t$ for p a prime number, and further to the case $q-1=\prod_{i=1}^{r} p_i^{e_i}$, where the p_i are the distinct prime divisors of $q-1$.

Using concepts from finite geometry we also show that these ideas naturally carry over to finite extension fields, and we give an algorithm for this case. The exploitation of the multiplicative subgroup structure in Moenck's generalized algorithm in $GF(q)$ is shown to be analogous to the exploitation of the affine subspace structure in the new algorithm developed for root finding in finite extension fields $GF(q^m)$.

Von zur Gathen [6] has shown that for primes p for which the prime factors of $p-1$ are small, the problem of factoring polynomials over finite fields **F** of characteristic p is polynomial-time equivalent to finding a primitive element modulo p. For such primes p, that paper presents an algorithm which, given as input a polynomial over **F** and a primitive element modulo p, achieves a factorization of the polynomial in deterministic polynomial time. Whereas von zur Gathen generalizes Moenck's result, we generalize Moenck's algorithm itself (and Moenck's result). Von zur Gathen's generalization is broader—he gives a factorization technique, whereas our algorithm, like Moenck's original algorithm, is restricted to root finding. Of course, factoring univariate polynomials over finite fields of characteristic p is polynomial-time reducible to factoring univariate square-free polynomials with only linear factors over $GF(p)$ [6, Thm. 2.4]. Von zur Gathen's algorithm finds a nontrivial factor of a polynomial $f(x)\in GF(p^l)[x]$ in $O(n^8)$ bit operations, using asymptotically fast integer and polynomial arithmetic, where n is the maximum of $\log p$, l, $\deg f(x)$, and the largest prime

* Received by the editors June 19, 1989; accepted for publication (in revised form) May 20, 1991.
† Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada.
‡ Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada. Present address, Bell-Northern Research, P.O. Box 3511, Station C, Ottawa, Canada, K1Y 4H7.

factor b of $p-1$. It employs a basis for the so-called Berlekamp subalgebra, and involves the computation of p_ith roots in $GF(p)$, where $p_i | p-1$. (It is interesting to note that in the computation of p_ith roots, discrete logarithms in $GF(p)$ are effectively computed using a technique popularized by Pohlig and Hellman [13], which is efficient when $p-1$ has only small prime factors.) Our algorithm (see running time given below) is conceptually simpler than that of von zur Gathen, but then again the root finding problem is conceptually simpler (although theoretically equivalent) than factoring.

In [15] Ronyai presents a deterministic polynomial-time algorithm for factoring polynomials over $GF(p)$, where $p-1 = T \cdot t^l$, and $t \approx T$. The algorithm is a theoretical improvement on the von zur Gathen method in the sense that a primitive element modulo p is not required. The algorithm does not provide a practical savings, however, as it is not very efficient, and since a primitive element of $GF(p)$ can be efficiently found probabilistically given the factorization of $p-1$. Shoup [16] has recently refined the algorithms of [6] and [15] and proven the existence of a deterministic algorithm for factoring polynomials over $GF(p)$, which runs in time $\sqrt{b}(n \log p)^{O(1)}$ under the Extended Riemann Hypothesis, where n is the degree of the polynomial and b is the largest prime divisor of $p-1$. (Note that both von zur Gathen's algorithm noted above, and ours discussed below, run in time polynomial in b.)

In this paper we describe a class of root finding algorithms which we refer to as "subgroup refinement algorithms." The first class of these algorithms (the generalized Moenck algorithm) exploits the multiplicative subgroup structure, while the second (the affine refinement method) uses the additive structure. The former is a special purpose algorithm in that (in order to be efficient) it requires $q-1$ to be smooth, and furthermore requires a primitive element in $GF(q)$. The second is a general purpose algorithm in extension fields $GF(q^m)$, but is only efficient when q is small. We find the paradigm of subgroup refinement to be useful in the root finding problem. The two new methods presented in this paper generalize and unify previous methods. Both methods are practical, provided that appropriate smoothness conditions hold. We summarize the running times of the new algorithms below.

THEOREM 1. *Let $f(x)$ be a degree n polynomial over $\mathbf{F} = GF(q)$ having n distinct roots in $\mathbf{F}$, and let $q-1$ be b-smooth. Then given a primitive element ω in $\mathbf{F}$, the generalized Moenck algorithm is deterministic and has running time $O(n^2 \log q(\log q + b))$ steps, where a step is a $GF(q)$ operation.*

THEOREM 2. *Let $f(x)$ be a degree n polynomial over $\mathbf{F} = GF(q^m)$ having n distinct roots in $\mathbf{F}$. The running time of the affine refinement method is $O(qmn(m+n^2))$ steps, where a step is a $GF(q^m)$ operation.*

The remainder of this paper is organized as follows. In § 2 we review Moenck's original algorithm for root finding. The generalization of Moenck's algorithm is presented in § 3, and its running time is given. The additive analogue of the generalized Moenck algorithm is given in § 4, and its running time and expected running time (Theorem 7) are studied in § 5.

2. A review of Moenck's root finding algorithm for special fields. Moenck's algorithm for finding roots in $GF(q)$ of polynomials over $GF(q)$ runs in deterministic polynomial time, given a primitive element ω for $GF(q)$ [10, § IV]. The algorithm is applicable in special fields $GF(q)$, where q is of the form

$$q-1 = T \cdot 2^t$$

with $T \approx t$. Here, as is our convention, q refers to a prime or prime power. (Note that while Moenck specified the algorithm in Z_p, it applies equally in $GF(q)$.) For convenience, we first review the algorithm.

Let $f(x)=\sum_{i=0}^{n}\lambda_i x^i$, $\lambda_i \in GF(q)$, be a monic polynomial which is the product of n distinct linear factors over $GF(q)$, and assume $f(0)\neq 0$ (i.e., $\lambda_0\neq 0$). Noting that

$$x^{q-1}-1=(x^{(q-1)/2}-1)(x^{(q-1)/2}+1),$$

those roots of $f(x)$ that are quadratic residues can be separated from those that are quadratic nonresidues via

$$f_1(x)=\gcd(f(x), x^{(q-1)/2}-1). \tag{2.1}$$

In general at stage i, for $1\leqq i\leqq t$, noting

$$x^{(q-1)/2^{i-1}}-1=(x^{(q-1)/2^i}-1)(x^{(q-1)/2^i}+1), \tag{2.2}$$

we may compute

$$f_i(x)=\gcd(f_{i-1}(x), x^{(q-1)/2^i}-1) \tag{2.3}$$

where $f_0(x)=f(x)$; $f_{i-1}(x)$ contains all roots of $f(x)$ that are $(q-1)/2^{i-1}$th roots of unity, and $f_i(x)$ those that are furthermore $(q-1)/2^i$th roots of unity. Proceeding in this manner eventually yields the polynomial $f_t(x)$ containing precisely those roots of $f(x)$ (if any) which are $(q-1)/2^t=T$th roots of unity. Now these roots are necessarily contained in the multiplicative group generated by $\omega^{(q-1)/T}$, where ω is any primitive element of $GF(q)$, and they can at this point be found by trial and error, using direct evaluation in $f(x)$. For T and t being $O(\log_2 q)$, the maximum number of elements to be so checked is about the same as the maximum depth of recursion t.

Those roots of $f(x)$ "left over" at each stage by (2.3), i.e., the roots of $g_i(x)=f_{i-1}(x)/f_i(x)$, must also be dealt with. These roots are $(q-1)/2^{i-1}$th roots of unity which are not $(q-1)/2^i$th roots of unity, and hence have the form

$$\zeta=(\omega^{2^{i-1}})^s$$

where s is odd. In order to proceed, suppose each such root ζ was multiplied by $\omega^{2^{i-1}}$ yielding

$$\xi=\zeta\cdot\omega^{2^{i-1}}=\omega^{s\cdot 2^{i-1}}\cdot\omega^{2^{i-1}}=(\omega^{2^i})^{(s+1)/2}.$$

Then, since 2 divides $s+1$, each such "shifted" root ξ would be a $(q-1)/2^i$th root of unity, and could be processed by the refinement technique outlined above; and each original root ζ could then be recovered once the corresponding root ξ was so found, by dividing ξ by $\omega^{2^{i-1}}$.

What remains to be specified is, given a polynomial $g_i(x)$ whose roots are of the form $\omega^{s_j\cdot 2^{i-1}}$ for various odd s_j, how to determine the polynomial $\overline{g_i}(x)$ with shifted roots $\omega^{s_j\cdot 2^{i-1}}\cdot\omega^{2^{i-1}}$. It is easily verified that if the roots of $h(x)=\sum_{j=0}^{l}\delta_j x^j$ are α_k, then the roots of $h_1(x)=\sum_{j=0}^{l} b^{l-j}\delta_j x^j$ are $b\alpha_k$. Hence

$$\overline{g_i}(x)=\prod_{j=1}^{l}\left(x-\omega^{s_j\cdot 2^{i-1}}\cdot\omega^{2^{i-1}}\right)=\sum_{j=0}^{l}(\omega^{2^{i-1}})^{l-j}\delta_j x^j$$

can be computed directly from the coefficients of $g_i(x)$ and l successive powers of $\omega^{2^{i-1}}$.

To specify the algorithm explicitly, let $Search(s(x), \omega^{2^t})$ be a subroutine that evaluates $s(x)$ at the T elements in the cyclic group generated by ω^{2^t}, returning the set of all roots of $s(x)$ so found, and let $Shift(s(x), \sigma)$ be a subroutine returning the polynomial $\bar{s}(x)$ whose roots are precisely those of $s(x)$ shifted by σ. The special form of $q-1$ (i.e., T and t) is assumed known, as is a primitive element ω for $GF(q)$. The algorithm is initially invoked as $Roots(f(x), \omega, 1)$, and returns the set of roots of $f(x)$.

Correctness of the algorithm follows from the discussion above. We let deg $h(x)$ denote the degree of $h(x)$.

Roots$(h(x), \gamma, i)$
1 if deg $h(x)=0$
2 RETURN($\emptyset$)
3 else if deg $h(x)=1$ [i.e., $h(x)=x+\lambda_0$]
4 RETURN($\{-\lambda_0\}$)
5 else if $i=t+1$
6 RETURN(*Search*$(h(x), \gamma)$)
7 else
8 $g_1(x):=\gcd(h(x), x^{(q-1)/2^i}-1)$
9 $g_2(x):=$ *Shift*$(h(x)/g_1(x), \gamma)$
10 RETURN(*Roots*$(g_1(x), \gamma^2, i+1)\cup$ *Roots*$(g_2(x), \gamma^2, i+1)/\gamma$)
11 endif

3. A generalization of Moenck's algorithm. We now give a generalization of Moenck's root finding algorithm. In the original algorithm, at the top level, roots are separated based on membership in the (multiplicative) subgroup of $GF(q)^*$ of index 2 or its coset (i.e., the quadratic residues and nonresidues). This subgroup is then successively refined by considering (within it) membership in the subgroup of index 2^2, then (within this latter subgroup) membership in the subgroup of index 2^3, and so forth until eventually the subgroup of index 2^t is exploited. At each stage, to facilitate further refinement, roots in the subgroup of index 2^i that are not elements of the subgroup of index 2^{i+1} are appropriately mapped ("shifted") into this latter subgroup.

We first consider the generalization from the case $q-1=T\cdot 2^t$ to the case where $q-1=T\cdot p^t$, for any prime p. The quadratic residues and nonresidues are replaced by the subgroup $C^{(p)}=\{\omega^{kp}: 0\leqq k<(q-1)/p\}$ of index p, and its cosets $w^jC^{(p)}$, $1\leqq j<p$; here again, ω is a primitive element for $GF(q)$. This subgroup of index p is then refined using the successively smaller subgroups of index p^i, $2\leqq i\leqq t$. Let \ denote set difference. Whereas for $p=2$ only a single "shift" is required to map the elements of $C^{(2^i)}\backslash C^{(2^{i+1})}$ onto $C^{(2^{i+1})}$, now at the ith stage a distinct "shift" is required to map each of the cosets of $C^{(p^{i+1})}$ within $C^{(p^i)}$ onto $C^{(p^{i+1})}$; we make use of the following result. We use the notation $a\,|\,b$ to stand for "a divides b."

LEMMA 3. *If A, B are positive integers such that $AB\,|\,(q-1)$, then*
(i) $x^{(q-1)/A}-1=\prod_{j=0}^{B-1}(x^{(q-1)/AB}-\sigma^j)$, *where* $\sigma=\omega^{(q-1)/B}$;
(ii) *the roots of* $c_j(x)=x^{(q-1)/AB}-\sigma^j$, $0\leqq j<B$ *are precisely the elements of the coset* $\omega^{jA}C^{(AB)}=\{w^{jA+kAB}: 0\leqq k<(q-1)/AB\}$; *and*
(iii) *for each j, $1\leqq j<B$, the mapping* $\Psi:\omega^{jA}C^{(AB)}\rightarrow C^{(AB)}$ *defined by* $\Psi(\beta)=\omega^{(B-j)A}\beta$ *maps the coset* $\omega^{jA}C^{(AB)}$ *onto the subgroup* $C^{(AB)}$.

Proof. To prove (ii), let $\beta\in\omega^{jA}C^{(AB)}$. Then $\beta=\omega^{jA+kAB}$ for some k, $0\leqq k<(q-1)/AB$, and

$$\beta^{(q-1)/AB}=(\omega^{(q-1)/B})^j\cdot(\omega^{q-1})^k=\sigma^j$$

so that $c_j(\beta)=\sigma^j-\sigma^j=0$. This characterizes all roots of $c_j(x)$, since $\deg c_j(x)=(q-1)/AB$.

To prove (i), note that by (ii), the set of roots of the product on the right consists of the union of all cosets of $C^{(AB)}$ in $C^{(A)}$, and hence consists of $C^{(A)}$, which is precisely the set of roots of the left-hand side.

Statement (iii) is immediate. □

The original algorithm, at stage i, used $A=2^{i-1}$, $B=2$, and required the single shift $\omega^{2^{i-1}}$ (cf. (2.2)). For $q-1=T\cdot p^t$, $1\leqq i\leqq t$, we may use $A=p^{i-1}$, $B=p$; and the jth coset may be shifted by $(\omega^{p^{i-1}})^{p-j}$. With this note, the modified algorithm for $q-1=T\cdot p^t$ can be given.

```
Roots1(h(x), γ, i)
1     if deg h(x) = 0
2          RETURN (∅)
3     else if deg h(x) = 1   [i.e., h(x) = x + λ_0]
4          RETURN ({-λ_0})
5     else if i = t + 1
6          RETURN (Search(h(x), γ))
7     else
7.1        for j from 0 to p - 1 do
           {
8              g(x) := gcd (h(x), x^{(q-1)/p^i} - σ^{p-j})
9              g_j(x) := Shift(g(x), γ^j)
           }
10         RETURN (∪_{j=0}^{p-1} Roots1(g_j(x), γ^p, i + 1)/γ^j)
11    endif
```

As before, the algorithm is initially invoked as $Roots1(f(x), \omega, 1)$. For use in line 8, $\sigma=\omega^{(q-1)/p}$ may be precomputed.

We now make the following observations. We have as an invariant $\gamma=\omega^{p^{i-1}}$, so that at line 9, $\gamma^j=\omega^{jp^{i-1}}$. Before line 7.1, we have the assertion $h(x)\,|\,(x^{(q-1)/p^{i-1}}-1)$. After line 9, the assertion $g_j(x)\,|\,(x^{(q-1)/p^i}-1)$ holds, since the roots of $g(x)$, which from line 8 and Lemma 3 are elements of $\omega^{(p-j)p^{i-1}}C^{(p^i)}$, are shifted by $\omega^{jp^{i-1}}$ in line 9. Correctness of the algorithm follows from these assertions.

The algorithm for $q-1=T\cdot p^t$ can be further extended when T is composite; the full generalization then applies to $GF(q)$ where $q-1=\prod_{i=1}^r p_i^{e_i}$, with the p_i being distinct primes, and say $p=p_1$ and $t=e_1$. If we define $b=\max_{1\leqq i\leqq r}\{p_i\}$, then $q-1$ is said to be *b-smooth*; the full generalization is then efficient if b is small with respect to $q-1$. To begin further refinements, the subgroup $C^{(p^t)}$ of order T is partitioned into a subgroup of (preferably prime) order s, where $s\,|\,T$ (say, $s=p_2$), and its cosets within $C^{(p^t)}$; as before, these cosets are then mapped to this subgroup. This corresponds to using Lemma 3 with $A=p^t$, $B=s$ giving

$$x^T-1=\prod_{j=0}^{s-1}(x^{T/s}-\sigma^j),\qquad \sigma=\omega^{(q-1)/s}.$$

Note here that the roots of $x^{T/s}-\sigma^j$ are elements of the coset $\omega^{jp^t}C^{(sp^t)}$, and a shift mapping this coset onto $C^{(sp^t)}$ is $\omega^{p^t(s-j)}$.

Subsequent refinements are now possible if T/s is composite, and so on. In this way the generalization can be completed. To this end, let $q-1=\prod_{i=1}^r p_i^{e_i}$, $e_i\geqq 1$, where the p_i are now ordered distinct primes with $p_1<p_2<\cdots<p_r$. Let $l=\sum_{i=1}^r e_i$, and let $LIST[1\cdots l]$ be a $1\times l$ array whose first e_1 entries are p_1, whose next e_2 entries are p_2, and so on, with the last e_r entries being p_r. The complete algorithm, which we formally call the *generalized Moenck algorithm* (or informally, *Roots2*), can then be specified as follows.

```
Roots2(h(x), γ, i, d)
1     if deg h(x) = 0
```

```
2        RETURN (∅)
3    else if deg h(x) = 1   [i.e., h(x) = x + λ_0]
4        RETURN ({−λ_0})
5    else if i = l
6        RETURN (Search(h(x), γ))
7    else
7.1     s := LIST[i]
7.2     d_i := d · s
7.3     σ := ω^((q−1)/2)
7.4     for j from 0 to s − 1 do
          {
8            g(x) := gcd (h(x), x^((q−1)/d_i) − σ^(s−j))
9            g_j(x) := Shift(g(x), γ^j)
          }
10       RETURN (∪_{j=0}^{s−1} Roots2(g_j(x), γ^s, i+1, d_i)/γ^j)
11   endif
```

To determine the roots of a polynomial $f(x)$ whose factors are all linear and distinct, the initial call is now $Roots2(f(x), \omega, 1, 1)$. The second formal parameter γ is a generator for the subgroup of $GF(q)^*$ currently being refined; the third formal parameter i specifies the location of the next prime to be used in the list $LIST[\ ]$ of prime factors of $q-1$; the fourth formal parameter d gives the index in $GF(q)^*$ of the subgroup generated by γ. Note that $d=\prod_{j=1}^{i-1} LIST[j]$ and $d_i=\prod_{j=1}^{i} LIST[j]$.

The correctness of this algorithm is now established.

THEOREM 4. *Let $f(x)$ be a degree n polynomial over $GF(q)$ having n distinct roots in $GF(q)$. The generalized Moenck algorithm computes precisely the roots of $f(x)$.*

Proof. As mentioned above, the initial invocation establishes parameters $h(x)=f(x)$, $\gamma=\omega$, $i=1$, $d=1$. We establish correctness by arguing (using recursion) that a call to the routine $Roots2(h(x), \gamma, i, d)$ returns precisely the roots of $h(x)$. We establish the following assertions:

(A1) prior to line 1: $\gamma=\omega^d$,

(A2) prior to line 1: $h(x)\mid x^{(q-1)/d}-1$.

Clearly both assertions hold for $i=1$. Note that (A2) says that all roots of $h(x)$ are in the subgroup generated by $\omega^d=\gamma$. For $i=1$, also note that the invocation to $Roots2$ performs according to claim for $\deg h(x)=0$ or 1. Next, consider line 5 and the case $i=l$ (possibly $l=1$, or otherwise). Then $Search(h(x), \gamma)$ evaluates $h(x)$ at the p_r elements in the cyclic group generated by γ, and is successful at finding the $\deg h(x)$ roots of $h(x)$, due to (A2). To analyze lines 7.1-9, we apply Lemma 3 with $A=d$ and $B=s$ (so $AB=ds=d_i$). By Lemma 3(i), the s gcds in line 8 partition the root set of $h(x)$ into the root sets of the s resulting polynomials $g(x)$ (where the root set of $g(x)=1$ is $\varnothing$) and for each j,

$$g(x)\mid x^{(q-1)/ds}-\sigma^{s-j}.$$

By Lemma 3(ii), all roots of $g(x)$ are in the coset $\omega^{(s-j)d}C^{(ds)}$. The shift by $\gamma^j=\omega^{jd}$ in line 9 maps these roots into the coset $C^{(ds)}$, by Lemma 3(iii). Hence past line 9, we have

$$g_j(x)\mid x^{(q-1)/d_i}-1.$$

This, together with the recursive calls on line 10, establish (A1) and (A2) for the recursive call of stage $i+1$. The division by γ^j corrects for the shift in line 9. The roots

of $h(x)$ are returned correctly from stage i, either directly (via lines 2, 4, or 6), or recursively from line 10. □

It should be clear that for $q-1=T\cdot p^t$, where T is prime, the generalized Moenck algorithm reduces to *Roots*1, and for $p=2$, *Roots*1 reduces to Moenck's original algorithm. Hence for $p=2$ and T prime, the new algorithm is the same as that given by Moenck, and generalizes Moenck's algorithm for $p\neq 2$ and/or composite T.

The running time of the algorithm, as given by Theorem 1 in § 1, is now established.

Proof of Theorem 1. Suppose that at stage i (i.e., recursion depth i), the nonlinear (shifted) factors of $f(x)$ found so far are $h_1(x),\cdots,h_k(x)$, of degrees $n_1,\cdots,n_k$ respectively. If $i=l$, where l is the number of not necessarily distinct prime factors of $q-1$ and is the maximum recursion depth, then with $q-1$ b-smooth, the roots of each $h_j(x)$ are separated by searching through a subgroup of order at most b. The total cost of this search is $O(bn_1)+\cdots+O(bn_k)=O(bn)$, since $\sum_{j=1}^{k} n_j\leqq n$. If $i<l$, then the polynomial $x^{(q-1)/d_i} \pmod{h_j(x)}$ for use in line 8 can be computed in $O(n_j^2\log q)$ steps using conventional polynomial arithmetic. For each invocation of *Roots*2 at stage i, at most b cosets are processed ($s\leqq b$ in line 7.4), and the cost of each gcd operation in line 8 is $O(n_j^2)$ steps. The total cost for stage i is thus

$$\sum_{j=1}^{k} O(n_j^2\log q)+b\sum_{j=1}^{k} O(n_j^2)=O(n^2(\log q+b)).$$

This leads to a running time for *Roots*2 of

$$O(bn+n^2 l(\log q+b))$$

steps, and note that $l\leqq\log_2 q$. □

COROLLARY 5. *If* $b=(\log q)^{O(1)}$, *then given a primitive element* ω, *the generalized Moenck algorithm is a deterministic root finding algorithm with running time polynomial in* n *and* $\log q$. □

Using known reductions for reducing the factoring problem to a root finding problem (e.g., see [6, § 2], [7, § 4.2]), this then yields an algorithm for general factorization of any polynomial $a(x)\in GF(q)[x]$ in time polynomial in $\deg a(x)$ and $\log q$, when $q-1$ is b-smooth.

4. A subspace analogue of the generalized Moenck algorithm. In this section we introduce an analogue of the generalized Moenck algorithm of the previous section, which can be used for root finding in fields $GF(q^m)$ where $m>1$ and q is a prime or prime power. The analogue is based on the affine method of root finding [17, § III], which is itself a generalization of Berlekamp's trace algorithm [2, § 5], and mirrors the successive refinement of a subgroup of $GF(q)^*$ in Moenck's algorithm by the refinement of linear subspaces of $AG(m,q)$, the m-dimensional affine geometry over $GF(q)$. As in the affine method, we will exploit the structure of affine polynomials to allow for more efficient computations. An affine polynomial over $GF(q^m)$ is a polynomial of the form $A(x)=a+\sum_{i=0}^{d} a_i x^{q^i}$, $a, a_i\in GF(q^m)$. For the necessary background on affine polynomials, the reader is referred to [7, § 3.4] or [17].

Let $f(x)=\sum_{i=0}^{n}\lambda_i x^i$, $\lambda_i\in GF(q^m)$, be a monic polynomial which is the product of n distinct linear factors over $GF(q^m)$, and let $F(x)=LAM[f(x)]$ denote the least affine multiple of $f(x)$, the monic affine polynomial over $GF(q^m)$ of smallest degree which $f(x)$ divides. Let $\{\alpha_1,\alpha_2,\cdots,\alpha_m\}$ be a fixed basis of $GF(q^m)$ over $GF(q)$, and let $T_d(x)=LAM[x\cdot\prod_{i=1}^{d}(x-\alpha_i)]$, $1\leqslant d\leqslant m$. The roots of the affine polynomial $T_d(x)$ are precisely the elements of $GF(q^m)$ that lie in the linear subspace V_d of dimension d, generated by $\{\alpha_1,\alpha_2,\cdots,\alpha_d\}$ over $GF(q)$. For convenience, we let $V_0=\{0\}$ and

$T_0(x)=x$. Noting that the elements of $GF(q^m)$ can be partitioned as

$$GF(q^m)=V_m=\bigcup_{\lambda\in GF(q)}(V_{m-1}+\lambda\alpha_m), \tag{4.1}$$

we have that

$$T_m(x)=x^{q^m}-x=\prod_{\lambda\in GF(q)}T_{m-1}(x-\lambda\alpha_m).$$

Hence the following sequence of greatest common divisors

$$G_\lambda(x)=\gcd(F(x),\ T_{m-1}(x-\lambda\alpha_m)),$$

$$g_\lambda(x)=\gcd(G_\lambda(x), f(x)),$$

with λ ranging over $GF(q)$, separates the roots of $f(x)$ according to membership in V_{m-1} or its (additive) cosets. This is done in two stages for computational efficiency (see [17]), taking advantage of the structure of affine polynomials. Generalizing (4.1), at stage i, for $1\leqq i\leqq m$, we have

$$V_{m-i+1}=\bigcup_{\lambda\in GF(q)}(V_{m-i}+\lambda\alpha_{m-i+1}), \tag{4.2}$$

and if $h(x)$ is a polynomial all of whose roots lie in V_{m-i+1}, then the sequence of greatest common divisors

$$G_\lambda(x)=\gcd(LAM[h(x)],\ T_{m-i}(x-\lambda\alpha_{m-i+1})),$$

$$g_\lambda(x)=\gcd(G_\lambda(x),\ h(x)),$$

with λ ranging over $GF(q)$, separates the roots of $h(x)$ according to membership in V_{m-i} or its cosets within V_{m-i+1}. Now, the roots of $g_\lambda(x)$ lie in the affine space $V_{m-i}+\lambda\alpha_{m-i+1}$, and these can be mapped into the linear subspace V_{m-i} by computing $g_\lambda(x+\lambda\alpha_{m-i+1})$. The refinement process may then be continued by separating the roots of these "shifted" polynomials, from which the original roots are then easily recovered. This eventually leads to the separation of the roots of $f(x)$ in at most m stages.

The complete algorithm is specified below, and is initially invoked as $Roots3(f(x), 1)$.

```
Roots3(h(x), i)
1   if deg h(x) = 0
2       RETURN (∅)
3   else if deg h(x) = 1   [i.e., h(x) = x + λ₀]
4       RETURN ({−λ₀})
5   else
6       H(x) := LAM[h(x)]
7       for λ ranging over GF(q) do
        {
8           G_λ(x) := gcd (H(x), T_{m−i}(x − λα_{m−i+1}))
9           g_λ(x) := gcd (G_λ(x), h(x))
10          ḡ_λ(x) := g_λ(x + λα_{m−i+1})
        }
11      RETURN (∪_{λ∈GF(q)} Roots3(ḡ_λ(x), i+1) + λα_{m−i+1})
12  endif
```

In order to compute the gcd in line 8, it is useful first to compute the affine residues $x^{q^j} \bmod H(x)$, $0 \leqq j \leqq m-1$. $G_\lambda(x)$ can then be computed as

$$G_\lambda(x) := \gcd(H(x), T_{m-i}(x - \lambda\alpha_{m-i+1}) \bmod H(x)).$$

The cost of computing these residues, and also the cost of computing $H(x)$ in line 6 in each invocation of *Roots*3, can be avoided by not performing the shift in line 10, but instead separating roots at stage i according to membership in V_{m-i} and all of its cosets within V_m. Hence at stage i, the polynomial representation of any of the q^i cosets of V_{m-i} may be needed. These are given by $T_{m-i}(x-\gamma)$, where γ ranges over the subspace generated by $\{\alpha_{m-i+1}, \alpha_{m-i+2}, \cdots, \alpha_m\}$ over $GF(q)$. Observe that $T_{m-i}(x-\gamma)$ is readily obtained from $T_{m-i}(x)$, since if $T_{m-i}(x) = \sum_{k=0}^{m-i} t_k x^{q^k}$, $t_k \in GF(q)$, then

$$T_{m-i}(x-\gamma) = T_{m-i}(x) - T_{m-i}(\gamma) = T_{m-i}(x) - \sum_{k=0}^{m-i} t_k \gamma^{q^k}. \tag{4.3}$$

The modified algorithm, which we formally call the *affine refinement method* (or informally, *Roots*4), is given below, and is initially invoked as *Roots*4$(f(x), 1, 0)$ (here $F(x) = LAM[f(x)]$). In *Roots*3$(h(x), i)$, all roots of $h(x)$ are known to lie in the subspace V_{m-i+1}, and the cosets of V_{m-i} in V_{m-i+1} were exploited; here in *Roots*4$(h(x), i, \gamma)$, all roots of $h(x)$ are known to lie in the coset $V_{m-i+1} + \gamma$, and the q cosets of V_{m-i} in V_m that are contained in $V_{m-i+1} + \gamma$ are exploited.

*Roots*4$(h(x), i, \gamma)$
1 if deg $h(x) = 0$
2 RETURN ($\varnothing$)
3 else if deg $h(x) = 1$ [i.e., $h(x) = x + \lambda_0$]
4 RETURN ($\{-\lambda_0\}$)
5 else
6 for λ ranging over $GF(q)$ do
{
7 $G_\lambda(x) := \gcd(F(x), T_{m-i}(x - \gamma - \lambda\alpha_{m-i+1}))$
8 $g_\lambda(x) := \gcd(G_\lambda(x), h(x))$
}
9 RETURN ($\bigcup_{\lambda \in GF(q)}$ *Roots*4$(g_\lambda(x), i+1, \gamma + \lambda\alpha_{m-i+1})$)
10 endif

Note that $F(x)$ and the residues $x^{q^i} \bmod F(x)$ used in the gcd computation in line 7, can now be precomputed and used in all invocations of *Roots*4. We remark that *Roots*2 can be optimized in the similar manner as *Roots*4 optimizes *Roots*3, by avoiding the shift in line 9 of *Roots*2. By doing this, the residues $x^{2^j} \bmod f(x)$, $0 \leqq j \leqq \log_2(q-1)$, can be precomputed and used in each invocation of *Roots*2 to reduce $x^{(q-1)/d_i}$ modulo $f(x)$ before doing the gcd in line 8.

We comment on the method of computing the affine polynomials $T_d(x)$, $1 \leqq d \leqq m-1$. An obvious method is to first compute $l(x) = x \cdot \prod_{i=1}^{d}(x - \alpha_i)$, and then the least affine multiple of $l(x)$ using a well-known technique (see [1, p. 245]). A more efficient scheme noted by Ore in [12] is to calculate iteratively:

$$T_1(x) = x^q - \alpha_1^{q-1} x$$

and

$$T_i(x) = [T_{i-1}(x)]^q - [T_{i-1}(\alpha_i)]^{q-1} T_{i-1}(x), \qquad 2 \leqq i \leqq m-1.$$

Finally, we note that the *Roots*3 and *Roots*4 algorithms partition the elements of $GF(q^m)$ at stage i according to the coefficient of α_{m-i+1} in their representations as linear combinations of the basis elements $\alpha_1, \alpha_2, \cdots, \alpha_m$. As described above, the algorithms are deterministic, and examine the coefficients of $\alpha_m, \alpha_{m-1}, \cdots, \alpha_1$, in that order.

Note that for any fixed ordering of basis elements, there will be polynomials $f(x)$, of degree as small as 2, containing pairs of roots that are separated only in the stage corresponding to the last basis element. In other words, there are specific inputs that force the algorithm into its worst case running time of m stages. To randomize the algorithm, we might use a random permutation $\sigma \in S_m$, and let $T_d(x) = LAM[x \prod_{i=1}^{d} (x - \alpha_{\sigma(i)})]$ for $1 \leqq d \leqq m$. Randomizing in this fashion results in an algorithm whose expected running time, averaged over all inputs, is as before, but which now does not favour/discriminate against any particular inputs; the worst case running time remains unchanged. This is a minor point, not to be dwelled on. We say more about the expected number of stages in the next section.

5. Analysis of the affine refinement method. In this section we establish the running time of the *Roots*4 algorithm, as given by Theorem 2 in § 1. As a basic step, we count **F**-multiplications, where $\mathbf{F} = GF(q^m)$.

Proof of Theorem 2. The cost of computing $F(x)$, the least affine multiple of $f(x)$, is $O(n^3)$ steps, where $n = \deg f(x)$; this is done only once. The affine residues $x^{q^j} \bmod F(x)$, $0 \leqq j \leqq m-1$ are computed by successive powerings of x by q, and this requires at most mn **F**-multiplications and the same number of **F**-exponentiations by q; this is also done only once. Having these residues, the affine polynomial $T_{m-i}(x)$ can be reduced modulo $F(x)$ in $O(mn)$ steps. The affine polynomial $T_{m-i}(x - \gamma - \lambda_{m-i+1})$ in line 7 of the algorithm can now be reduced modulo $F(x)$ in $O(m)$ steps by the observation made in (4.3), following which $G_\lambda(x)$ is obtained by an *affine* gcd operation in $O(n^2)$ steps. An affine gcd operation is similar to the gcd operation of ordinary polynomials. Briefly, if $A(x)$ and $B(x)$ are affine polynomials, then the remainder $R(x)$ of $A(x)$ divided by $B(x)$ is obtained using long division, except that at each stage of the division, a scalar multiple of the appropriate qth power of $B(x)$ is subtracted from the current remainder $R_i(x)$, so as to eliminate the leading term of $R_i(x)$. This leaves the affine structure of $R_i(x)$ intact. Finally, $g_\lambda(x)$ can then be computed in $O(n^2)$ steps, by first reducing the affine polynomial $G_\lambda(x)$ modulo $f(x)$ (using the residues $x^{q^j} \bmod f(x)$ that are needed to compute $F(x)$) and then performing the gcd operation. For any i, $1 \leqq i \leqq m$, there are at most n invocations of *Roots*4 with second input parameter equal to i. This leads to a running time for the affine refinement method of $O(qln(m+n^2))$, where l is the number of stages of the algorithm required to separate all the roots of $f(x)$. Finally, note that $l \leqq m$. □

We note that since the affine polynomials $T_d(x)$, $1 \leqq d \leqq m-1$ can be precomputed and reused for any root finding problem in a given field, the cost of computing them is not included in the overall cost.

COROLLARY 6. *For q fixed, the affine refinement method is a deterministic root finding algorithm, with running time polynomial in n and m.*

The running time of the affine method [17], for finding all the roots of a degree n polynomial over $GF(q^m)$ is $O(n^3 + ln(m+qn))$, where l is the number of stages of the algorithm required, and $l \leqq m$. Both the affine refinement method and the affine method are useful in practice when q is small. For the case q small, the running time

of the affine method is $O(n^3+ln(m+n))$, which is better than $O(ln(m+n^2))$, the running time of the affine refinement method. If, furthermore, $n^2<m$, then the running time of both methods is $O(lmn)$. A comparison of the affine method and Berlekamp's trace algorithm can be found in [17, § IV]. For q small, the running time of the trace algorithm is $O(mn^2+ln(m+n))$, and thus the affine method is faster for $\deg f(x)=n<m$.

We now calculate the expected number $E(l)$ of stages in *Roots*4 required to separate all the roots of a polynomial $f(x)$ of degree n over $GF(q^m)$, assuming that the roots of $f(x)$ are random and distinct. Let $P(i)$ denote the probability that all roots are separated in at most i stages. Clearly, $P(i)=1$ for $i\geqq m$. For $i<m$, $P(i)$ is then the probability that n randomly chosen and distinct elements of $GF(q^m)$ lie in distinct cosets of V_{m-i} within V_m. Thus,

$$P(i)=\frac{q^m(q^m-q^{m-i})\cdots(q^m-(n-1)q^{m-i})}{q^m(q^m-1)\cdots(q^m-(n-1))}$$

$$=\frac{q^{(m-i)n}\binom{q^i}{n}}{\binom{q^m}{n}}.$$

So, the probability that the n roots of $f(x)$ are separated in exactly i stages is $P(i)-P(i-1)$. Whence,

$$E(l)=\sum_{i=1}^{\infty} i[P(i)-P(i-1)]$$

$$=m-\sum_{i=1}^{m-1} P(i).$$

We note that $E(l)$ is also the expected number of stages required to separate all the roots of $f(x)$ in Berlekamp's trace algorithm, as well as in the affine method. A bound for $E(l)$ is $E(l)\leqq 2\log_q m+O(1)$. This bound is derived, for example, in [8, p. 106], where the author is concerned with the expected depth of a $q$ary compressed TRIE for a set of m elements.

THEOREM 7. *The expected number of stages required to separate all the roots of $f(x)$ in the affine refinement method, the affine method, and Berlekamp's trace method is at most* $2\log_q m+O(1)$. *The expected running times of the affine refinement method, the affine method, and Berlekamp's trace method are* $O(qn\log_q m(m+n^2))$, $O(n^3+n\log_q m(m+qn))$, *and* $O(mn^2+n\log_q m(m+qn))$ *steps, respectively.*

As in the case of Berlekamp's trace method and the affine method, the use of optimal normal bases (see [11]) enhances the performance of the affine refinement method in $GF(p^m)$, for p a prime. The affine refinement method, for fields of characteristic 2, was implemented in the *C* programming language on a *SUN* 3/160 computer, making use of the supporting routines used to implement the trace and affine methods in [9]. The affine refinement method was found to be slightly slower in practice than the affine method, and this is due to the overhead of computing the polynomial $T_{m-i}(x-\gamma-\lambda\alpha_{m-i+1})$ from $T_{m-i}(x)$, and also the gcd of affine polynomials in line 7 of *Roots*4.

Acknowledgments. The authors thank the referees for their suggestions, which considerably improved the presentation of this work, and for pointing out references

[15] and [16]. We would also like to thank one of the referees for indicating an error in a preliminary version of Theorem 2.

REFERENCES

[1] E. R. BERLEKAMP, *Algebraic Coding Theory*, McGraw-Hill, New York, 1968.

[2] ———, *Factoring polynomials over large finite fields*, Math. Comp., 24 (1970), pp. 713-735.

[3] P. CAMION, *A deterministic algorithm for factoring polynomials of $F_q[x]$*, Ann. Discrete Math., 17 (1983), pp. 149-157.

[4] ———, *Improving an algorithm for factoring polynomials over a finite field and constructing large irreducible polynomials*, IEEE Trans. Inform. Theory, 29 (1983), pp. 378-385.

[5] D. G. CANTOR AND H. ZASSENHAUS, *A new algorithm for factoring polynomials over finite fields*, Math. Comp., 36 (1981), pp. 587-592.

[6] J. VON ZUR GATHEN, *Factoring polynomials and primitive elements for special primes*, Theoret. Comput. Sci., 52 (1987), pp. 77-89.

[7] R. LIDL AND H. NEIDERREITER, *Finite Fields*, Cambridge University Press, Cambridge, UK, 1987.

[8] K. MEHLHORN, *Data Structures and Algorithms* 1: *Sorting and Searching*, Springer-Verlag, Berlin, 1984.

[9] A. J. MENEZES, P. C. VAN OORSCHOT, AND S. A. VANSTONE, *Some computational aspects of root finding in $GF(q^m)$*, in Symbolic and Algebraic Computation, Lecture Notes in Computer Science 358 Springer-Verlag, New York, 1989, pp. 259-270.

[10] R. T. MOENCK, *On the efficiency of algorithms for polynomial factoring*, Math. Comp., 31 (1977), pp. 235-250.

[11] R. C. MULLIN, I. M. ONYSZCHUK, S. A. VANSTONE, AND R. M. WILSON, *Optimal normal bases in $GF(p^n)$*, Discrete Appl. Math., 22 (1988/89), pp. 149-161.

[12] O. ORE, *On a special class of polynomials*, Trans. Amer. Math. Soc., 35 (1933), pp. 559-584.

[13] S. C. POHLIG AND M. E. HELLMAN, *An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance*, IEEE Trans. Inform. Theory, 24 (1978), pp. 106-110.

[14] M. O. RABIN, *Probabilistic algorithms in finite fields*, SIAM J. Comput., 9 (1980), pp. 273-280.

[15] L. RONYAI, *Factoring polynomials modulo special primes*, Combinatorica, 9 (1989), pp. 199-206.

[16] V. SHOUP, *Smoothness and factoring polynomials over finite fields*, Inform. Process. Lett., 38 (1991), pp. 39-42.

[17] P. C. VAN OORSCHOT AND S. A. VANSTONE, *A geometric approach to root finding in $GF(q^m)$*, IEEE Trans. Inform. Theory, 35 (1989), pp. 444-453.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 240–266, April 1992
© 1992 Society for Industrial and Applied Mathematics
004

LEARNING INTEGER LATTICES*

DAVID HELMBOLD†, ROBERT SLOAN‡, AND MANFRED K. WARMUTH†

Abstract. The problem of learning an integer lattice of $\mathbf{Z}^k$ in an on-line fashion is considered. That is, the learning algorithm is given a sequence of k-tuples of integers and predicts for each tuple in the sequence whether it lies in a hidden target lattice of $\mathbf{Z}^k$. The goal of the algorithm is to minimize the number of prediction mistakes. An efficient learning algorithm with an absolute mistake bound of $k + \lfloor k \log(n\sqrt{k})\rfloor$ is given, where n is the maximum component of any tuple seen. It is shown that this bound is approximately a $\log\log n$ factor larger than the lower bound on the worst case number of mistakes given by the VC dimension of lattices that are restricted to $\{-n, \cdots, 0, \cdots, n\}^k$.

This algorithm is used to learn rational lattices, cosets of lattices, an on-line word problem for abelian groups, and a subclass of the commutative regular languages. Furthermore, by adapting the results of [D. Helmbold, R. Sloan, and M. K. Warmuth, *Machine Learning*, 5(1990), pp. 165–196], one can efficiently learn nested differences of each of the above classes (e.g., concepts of the form $c_1 - (c_2 - (c_3 - (c_4 - c_5)))$, where each c_i is the coset of a lattice).

Key words. lattices, concept learning, mistake bounds

AMS(MOS) subject classifications. 68T05, 68Q25, 06B99

1. Introduction.

Integer lattices are one of the most basic combinatorial structures. An integer lattice is a nonempty set of k-tuples from $\mathbf{Z}^k$ that is closed under addition and subtraction. Let $\mathcal{L}^k$ be the concept class consisting of all integer lattices in $\mathbf{Z}^k$. In this paper we present an algorithm for learning $\mathcal{L}^k$, and prove that its performance is within a log log factor of optimal in the on-line model of learning, using the worst case number of mistakes as the performance criterion. Note that any learning algorithm with a good on-line mistake bound can be used as a subroutine to construct a PAC learning algorithm [17], [4], but some PAC learning algorithms have very poor mistake bounds.

Although the learning algorithm is of interest itself, this paper's major technical contributions are analyzing the learning performance of that algorithm and computing the VC dimension for the class of integer lattices (thus giving a lower bound on the worst case number of mistakes made by any learning algorithm). In particular, we prove that our learning algorithm has an absolute mistake bound of $k(1+\log(n\sqrt{k}))$,[1] where n is an upper bound on the absolute value of any component of any instance seen, and that no learning algorithm can have a mistake bound of less than $(1-\epsilon)k\ln n/\ln\ln n$ for any $\epsilon > 0$. Thus we achieve nearly optimal learning performance in a very strict model.[2]

* Received by the editors July 30, 1990; accepted for publication January 15, 1991.

† Department of Computer and Information Sciences, University of California, Santa Cruz, California 95060. The research of the first author was supported in part by a Regents' Junior Faculty fellowship and by Office of Naval Research grant N00014-85-K-0554. The research of the third author was supported by Office of Naval Research grant N00014-86-K-0454. Part of this work was done while the third author was visiting Aiken Computation Laboratory, Harvard University, and was supported by Office of Naval Research grant N00014-85-K-0554.

‡ Electrical Engineering and Computer Science Department, Box 4348, University of Illinois, Chicago, Illinois 60680. Much of this research was done while this author was visiting the Department of Computer and Information Sciences, University of California, Santa Cruz, and was supported by Office of Naval Research grant N00014-86-K-0454. This author's research was also supported by United States Army Research Office grant DAAL 03-86-K-0171 and by a National Science Foundation Graduate Fellowship.

[1] Throughout, log represents the base two logarithm, and ln represents the natural logarithm.

[2] Abe [1] considers learning the harder class of semilinear sets. Using different parameters, he presents a learning algorithm for semilinear sets when $k = 2$.

The algorithm we present keeps a basis for the "smallest" lattice containing all positive examples seen so far and predicts on new instances according to whether they are in this lattice or not. Although any reasonable basis can be used, our algorithm keeps a special basis which facilitates prediction and updates while storing only a small number of derived examples.

Our algorithm is similar to the algorithms of Kannan and Bachem [15] and Chou and Collins [25] for converting an integer basis into a special integer basis called a Hermite normal form (HNF). However, in our application we are not given all the vectors (positive instances) in advance, so we must convert the HNF algorithm into an on-line algorithm. Significant adaptations of the HNF algorithm and its analysis were required for our application, since we want to keep a HNF-like basis while efficiently processing new examples.

We also present several applications of the learning algorithm for $\mathcal{L}^k$. For example, it can be used to learn rational lattices, cosets of lattices, and an on-line word problem for abelian groups. Furthermore, we can use it to learn the subclass of commutative regular languages accepted by DFAs whose single final state reaches the start state. This subclass includes the class of "counter languages." The last result is surprising, because the counter languages are precisely the regular languages used to prove that the minimum consistent DFA problem for regular languages cannot be approximated within any polynomial [21] (see §6.5 for details). Finally, adapting the results from a companion paper [14], our algorithm can be applied to efficiently learn nested differences of all the above learnable classes.

2. Methods and outline.

The setting we will be concerned with is that of on-line learning. Formally, *concepts* are subsets of some *instance space* X from which instances are drawn and a *concept class* is a subset of 2^X, the power set over X. The instances are labeled *consistently* with a fixed *target concept* c which is in the *concept class* C to be learned; i.e., an instance is labeled "+" if it lies in the target concept and "−" otherwise. Labeled instances are called *examples.* An *on-line learning algorithm* interactively participates in a series of *trials.* On each trial, the algorithm gets an instance and predicts what its label is. After predicting, the on-line algorithm is informed of the instance's true label. A *mistake* is a trial where the on-line algorithm makes an incorrect prediction. Between trials, the algorithm's *hypothesis* is the subset of the instance space where the algorithm predicts "+."

2.1. Example: Learning vector subspaces.

Imagine for the time being that our instance space is an arbitrary vector space $\mathcal{S}$, and that our concept class is the set of all vector subspaces of $\mathcal{S}$. A natural on-line learning algorithm for this problem, shown to us by Haussler and inspired by a similar algorithm of Shvaytser [23] is described in Fig. 1.

2.2. Algorithm V is a closure algorithm.

The class of all subspaces of a vector space is intersection-closed, and Algorithm V is in fact a special case of an algorithm for learning intersection-closed concept classes called the "closure algorithm" [20], [6], [14].

DEFINITION. A concept class is *intersection-closed* if for any finite set contained in some concept, the intersection of all concepts containing the finite set is also a concept in the class.

DEFINITION. Fix an intersection-closed concept class $\mathcal{C}$ on some instance space. Let S be a set of positive examples of some concept from $\mathcal{C}$. The *closure* of S with respect to the concept class $\mathcal{C}$ (written CLOSURE(S) when $\mathcal{C}$ is understood) is the

Algorithm V

Predict that the zero vector is positive and all other vectors are negative until a mistake is made.
Whenever a mistake is made predicting the label of some instance, add that instance to a hypothesized basis set for the target subspace.
In general, predict that any instance in the span of the hypothesized basis is positive, and all other instances are negative.

FIG. 1. *An algorithm for learning subspaces of a vector space.*

intersection of all concepts in $\mathcal{C}$ containing the instances of S.

Thus the closure of a set of instances is the smallest concept containing all those instances. The *closure algorithm* is a generic on-line learning algorithm whose hypothesis is CLOSURE(POS) where POS is the set of positive examples seen so far. If the instance space is a vector space, and the concept class is the set of all subspaces, then the closure of a set of positive examples (vectors) is their span. Thus Algorithm V is a closure algorithm.

2.3. How good is closure algorithm V for learning subspaces? This paper uses the mistake-based performance criteria of Littlestone [16] to measure the performance of on-line learning algorithms. For any learning algorithm Q and target concept c define $M_Q(c)$ to be the maximum number of mistakes that Q makes on any possible sequence of instances labeled according to c. For any nonempty concept class C, define $M_Q(C) = \max_{c \in C} M_Q(c)$. Any bound on $M_Q(C)$ is called an (*absolute*) *mistake bound* for algorithm Q applied to class C. The *optimal mistake bound* for concept class C, *opt*(C), is the minimum over all learning algorithms Q of $M_Q(C)$.

One lower bound on the mistake bound of any learning algorithm for a concept class is given by a very useful combinatorial parameter, the Vapnik–Chervonenkis (VC) dimension [24].

DEFINITION. A set of instances S is *shattered* (by the concept class C) if for each subset $S' \subseteq S$, there is a concept $c \in C$ that contains all of S', but none of the instances in $S - S'$. The *Vapnik–Chervonenkis dimension* of concept class C, denoted by VCdim(C), is the cardinality of the largest set shattered by the concept class.

Littlestone [16] has given the following relationship between the VCdim(C) and the optimal mistake bound.

THEOREM 2.1. *For every concept class C,* VCdim$(C) \leq$ *opt*(C).

It is fairly easy to see that the closure algorithm V, has an absolute mistake bound of the vector space dimension of $\mathcal{S}$ when learning subspaces of vector space $\mathcal{S}$. Each time a mistake is made, a new vector is added to the basis set Algorithm V maintains, and this can happen at most as many times as the vector space dimension of $\mathcal{S}$. The VC dimension of this concept class is also the vector space dimension of $\mathcal{S}$, so in this case the closure algorithm is optimal.

2.4. The closure algorithm is not always good. Even if a concept class is not intersection-closed, it can always be embedded in one that is. Thus the closure algorithm applies to any concept class. However, there are a number of potential problems:

1. There might not be an efficient algorithm for computing the closure of a set of instances.

2. Given a representation of the closure, it might be difficult to predict whether a new instance is in the closure.
3. The closure algorithm might not have a good mistake bound.

For example, regular sets are intersection-closed. The closure of a finite set of words equals the set itself and thus the closure algorithm makes a mistake on each new positive instance. So here the first two problems do not arise, but the third problem does. If we restrict the concept class to regular sets representable by DFAs with at most s states, then the class is no longer intersection-closed.

Theorem 2.1 implies that any algorithm makes at least as many mistakes as the VC dimension. However, even when the concept class is intersection-closed and has a small VC dimension, the closure algorithm can still make a large number of mistakes. For example, if the instance space is $0, \cdots, n$ and the concepts are initial segments (i.e., the intervals $[0, i]$ for $1 \leq i \leq n$), then the mistake bound of the closure algorithm is n (consider increasing sequences of positive examples) even though the concept class has VC dimension 1.

2.5. Applying the closure algorithm to integer lattices. As in the case of vector spaces, the concept class $\mathcal{L}^k$ of integer lattices of $\mathbf{Z}^k$ is intersection-closed. For any set $S \subseteq \mathbf{Z}^k$, the closure of S is the set of all k-tuples produced by summing integer multiples of elements in S. Thus the generic closure algorithm can be used to learn $\mathcal{L}^k$. However, lattices are significantly more complicated than vector spaces. Any implementation of the closure algorithm must take into account the "holes" in the lattice. For example, consider the lattice generated by the basis $(2,2)$ and $(0,2)$. Although the point $(1,2)$ can be written as $\frac{1}{2}(2,2)+\frac{1}{2}(0,2)$, it is not in the lattice, as it is not an *integer* linear combination of the basis vectors. Determining if a point is in a lattice involves solving a set of linear Diophantine equations rather than inverting a matrix.

We give a particular implementation of the closure algorithm, called Algorithm A, which overcomes the three potential problems stated in the previous section.

1. Algorithm A (presented in the Appendix) efficiently computes closures with respect to $\mathcal{L}^k$.
2. Algorithm A represents closures so that prediction can be done efficiently.
3. We prove a good mistake bound for the closure algorithm when applied to lattices.

We show in §3 below that $\text{VCdim}(\mathcal{L}^k)$ is infinite. Thus we know by Theorem 2.1 that the mistake bound of any algorithm must be infinite. To overcome this difficulty, we restrict our attention to $\mathcal{L}^k(n)$, which we define to be the class $\mathcal{L}^k$ where instances are restricted to $\{-n, \cdots, 0, \cdots, n\}^k$. The concept class $\mathcal{L}^k(n)$ is also intersection-closed, as is the restriction of any intersection-closed class to some subset of its domain. Theorem 3.3 below shows that $\text{VCdim}(\mathcal{L}^k(n))$ is roughly $k \ln n / \ln\ln n$.

In this paper we prove that the absolute mistake bound of the closure algorithm when learning $\mathcal{L}^k(n)$, $M_{\text{CLOSURE}}(\mathcal{L}^k(n))$, is at most $k + \lfloor k \log(n\sqrt{k}) \rfloor$. This bound is only a factor of roughly $\log\log n$ larger than the lower bound on the worst case number of mistakes given by the VC dimension of $\mathcal{L}^k(n)$.

Remark. We will not actually limit the size of the input to the closure algorithm; we will simply be describing its performance as a function of the size of its inputs.

2.6. Our algorithm and its advantages. Algorithm A (fully described in the Appendix) for learning $\mathcal{L}^k(n)$ is a particular implementation of the closure algorithm since its hypothesis is always the closure of the positive examples seen so far. Thus Algorithm A has the same mistake bound as the generic closure algorithm. Our

algorithm, however, has some space and computational advantages over the obvious implementations of the closure algorithm.

One obvious implementation of the closure algorithm saves all previous mistakes. After each mistake it recomputes a lower triangular basis for the smallest lattice containing the positive examples using the HNF algorithms of [15], [25]. Prediction is done by back substitution in this lower triangular matrix. Note that the number of mistakes can be at least as large as the VC dimension. As we shall see in §3, the VC dimension is bounded below by approximately $k \ln n / \ln \ln n$, thus this obvious implementation of the closure algorithm requires storing about $k \ln n / \ln \ln n$ positive examples.

In contrast, Algorithm A stores only the lower triangular basis (k derived positive examples). Rather than recomputing the basis from scratch after each mistake, it is updated on-line. The analysis of Algorithm A is nontrivial and is included in the Appendix. One of the more difficult parts is bounding the number of bits required to represent a derived example. The other contribution of this paper is the application of Algorithm A to the problems described in §6.

2.7. Outline of the paper. In the following section we compute the VC dimension of $\mathcal{L}^1(n)$, lattices in one dimension, and bound the VC dimension of $\mathcal{L}^k(n)$. In §4 we compute lower bounds on $opt(\mathcal{L}^k(n))$, the minimum mistake bound any algorithm can achieve when learning $\mathcal{L}^k(n)$. The closure algorithm's mistake bound is then analyzed in §5. At the end of the section, we show that a modified version of the halving algorithm [5], [19], [4] has a nearly optimal mistake bound when learning $\mathcal{L}^1(n)$. Section 6 shows how Algorithm A can be extended to learn rational lattices, cosets of lattices, and a word problem for abelian groups. We conclude §6 by showing how Algorithm A can be applied to learning a subclass of commutative regular languages. In §7 we discuss how Algorithm A can be used in conjunction with a master algorithm for learning nested differences. Our master algorithm is a modification of a similar algorithm presented in a companion paper [14]. It leads to efficient learning algorithms for nested differences of any of the concept classes that we learn using Algorithm A. A short summary of our results is given in the concluding section. In the Appendix we formally state Algorithm A and prove bounds on its resource requirements.

3. VC dimension of integer lattices. This section contains bounds on the VC dimension of the concept class $\mathcal{L}^k(n)$. The VC dimension provides a lower bound on both the number of examples stored by the standard implementation of the closure algorithm [14], and on the mistake bound of any learning algorithm.

We begin by exactly calculating the VC dimension of $\mathcal{L}^1(n)$. Note that each concept in $\mathcal{L}^1(n)$ can be represented by an integer between 0 and n. The concept represented by j contains the subset of $-n, \cdots, n$ whose members are multiples of j.

THEOREM 3.1. *For all* $n \geq 1$,

$$\mathrm{VCdim}(\mathcal{L}^1(n)) = \max\{r \mid 2 \cdot 3 \cdot 5 \cdots p_r \leq 2n\},$$

where p_i *is the ith prime.*

We start with a definition we will need in our proof.

DEFINITION. Let S be any shattered set; let $T \subseteq S$. We call a concept c such that $T = c \cap S$ a *witness* for T.

Proof. Let r be maximum such that $P = \prod_{i=1}^{r} p_i \leq 2n$ and $d = \mathrm{VCdim}(\mathcal{L}^1(n))$. Now we show $d \geq r$ by exhibiting set S with $|S| = r$ that is shattered: S is all

products of all but one of the first r primes. In symbols, $S = \{P/p_i \mid 1 \le i \le r\}$. Since $P/p \le n$ for all primes p, every element of S is in the instance space. For every $x = P/p_i$ in S, define $\hat{x} = p_i$. It is easy to see that S is shattered: The witness for any nonempty $T \subseteq S$ is $P/\prod_{x \in T} \hat{x} = \prod_{x \notin T} \hat{x}$. The witness for S is 1, and the witness for the empty set is 0.

Hence $d \ge r$. Now we show that $d \le r$.

Let $S = \{x_1, x_2, \cdots, x_d\}$ be a largest shattered set. First we argue that we may assume that there is no $s > 1$ that divides all elements of S. If there is such an s, we can work instead with $S' = \{x_1/s, x_2/s, \cdots, x_d/s\}$ and divide each witness by s.

Call any subset of $d-1$ elements of S a *minor*. Set S has d minors, S_1 through S_d (where S_i is the minor *not* containing x_i). Let t_i be a witness for S_i.

Now no t_i can be 1, since $t_i \not| x_i$. (Note that $0 \notin S$, because 0 is a positive example of every concept and S is shattered.) Furthermore, $\gcd(t_i, t_j) = 1$ for all $i \ne j$, since $\gcd(t_i, t_j) \mid x$ for every $x \in S$, and we assumed that 1 is the only number with this property. Thus it must be that for each i there is a prime p_i such that $p_i \mid t_i$ but $p_i \not| t_j$ for any $j \ne i$.

Pick any odd $x \in S$. Element x is in $d-1$ minors of S so $d-1$ different t_i's divide x, and x is a multiple of at least $d-1$ different p_i's. Since $x \le n$ and $2 \not| x$, there are d distinct primes ("2" plus the $d-1$ primes dividing x) whose product is at most $2n$, thus $d \le r$. □

In order to obtain numerical bounds on the VC dimension, we recall some facts from number theory (see, e.g., Hardy and Wright [13, pp. 262–263]):

1. Let $f(n)$ be the maximum number of consecutive primes such that $\prod_{i=1}^{f(n)} p_i \le n$. Then for every $\epsilon > 0$, for all sufficiently large n we have

$$f(n) > \frac{(1-\epsilon)\ln n}{\ln \ln n}. \tag{1}$$

2. For all $\epsilon > 0$, for all sufficiently large m, we have

$$\log(\tau(m)) < \frac{(1+\epsilon)\ln m}{\ln \ln m}, \tag{2}$$

where $\tau(m)$ is the number of positive divisors of m.

COROLLARY 3.2. *For all ϵ, for sufficiently large n,*

$$\frac{(1-\epsilon)\ln n}{\ln \ln n} < \mathrm{VCdim}(\mathcal{L}^1(n)) < \frac{(1+\epsilon)\ln n}{\ln \ln n}.$$

Proof. The left inequality follows directly from Theorem 3.1 together with (1), once we note that $\ln n/\ln\ln n < \ln 2n/\ln\ln 2n$.

For the right inequality, we need to bound $f(2n)$, where f is the function specified in (1). Let m be a particular integer of the form $m = 2 \cdot 3 \cdot 5 \cdots p_r$. Note that for such an m we have $\log(\tau(m)) = f(m)$, and thus $\max_{m \le n} \log(\tau(m)) \ge f(n)$. Thus we have an upper bound on $f(n)$ in terms of the log of the number of divisors of any $m \le n$, and can apply (2) to get the desired result. □

Remark. The preceding should make it clear that $\mathrm{VCdim}(\mathcal{L}^1(n))$ can be made arbitrarily large by choosing a suitable value for n, and thus that $\mathrm{VCdim}(\mathcal{L}^1)$ is infinite.

We now get a lower bound on the VC dimension of the more general concept class $\mathcal{L}^k(n)$.

THEOREM 3.3.

$$k + \left\lfloor k \log(n\sqrt{k}) \right\rfloor \geq \mathrm{VCdim}(\mathcal{L}^k(n)) \geq k\mathrm{VCdim}(\mathcal{L}^1(n)).$$

Proof. The first inequality is Corollary 5.3, proven later. The proof of the second inequality is essentially a particular case of a bound of Dudley's on the VC dimension of cross products of those concept classes, among them $\mathcal{L}^1(n)$, that have a certain property he calls "being bordered" [9, Thm. 9.2.14].

Let S be the exhibited set of numbers shattered in the proof of Theorem 3.1. Let U be the set of size $k\,|S|$ consisting of all k-tuples of integers containing $k-1$ zeros and one value from S. In this case witnesses are sets of k-tuples. To shatter any $T \subseteq U$, first write $T = T_1 \cup T_2 \cup \cdots \cup T_k$, where all elements of T_i have nonzero values only in position i. For each T_i, let t_i be the number that is the witness for the set of all nonzero components of all elements of T_i (viewed as a concept from $\mathcal{L}^1(n)$ as in the proof of Theorem 3.1). The witness for T is then the set of all integer combinations of the k tuples that have t_i in the ith position and 0's elsewhere. □

The lower bound in Theorem 3.3 is not tight. For example, $\mathrm{VCdim}(\mathcal{L}^1(2)) = 1$, but $\mathrm{VCdim}(\mathcal{L}^2(2)) = 3$ since $(1, 2)$, $(2, 1)$, and $(2, 2)$ are shattered. Determining tight bounds on $\mathrm{VCdim}(\mathcal{L}^k(n))$ remains an open problem.

4. Lower bounds on learning $\mathcal{L}^k(n)$. We know from Theorem 2.1 that $opt(\mathcal{L}^k(n)) \geq \mathrm{VCdim}(\mathcal{L}^k(n))$, so from Theorem 3.3 together with Corollary 3.2, we get a first lower bound for $opt(\mathcal{L}^k(n))$.

COROLLARY 4.1. *For all $\epsilon > 0$, for sufficiently large n,*

$$opt(\mathcal{L}^k(n)) \geq k\,(1-\epsilon)\,\frac{\ln n}{\ln \ln n}.$$

For sufficiently large n we can get a slightly better lower bound on $opt(\mathcal{L}^1(n))$ than $\mathrm{VCdim}(\mathcal{L}^1(n))$ using an adversary argument.[3]

THEOREM 4.2.

$$opt(\mathcal{L}^1(n)) \geq \max_{m \leq n} \sum_{i=1}^{s} \lfloor \log(e_i + 1) \rfloor$$

where $m = \prod_{i=1}^{s} p_i^{e_i}$.

Proof. Let $m = \prod p_i^{e_i}$ be a number less than or equal to n with a maximal number of divisors. Our adversary begins by first giving m as a positive instance. The adversary then makes the algorithm perform a search similar to binary search for the value of each exponent as follows. The next group of instances begin with $p_1^{\lfloor e_1/2 \rfloor} \prod_{i \geq 2} p_i^{e_i}$, and continues with various exponents for p_1 (times $\prod_{i \geq 2} p_i^{e_i}$). The algorithm is forced to make $\lfloor \log(e_1 + 1) \rfloor$ mistakes since there are $e_1 + 1$ choices for the exponent of p_1. The following group of instances all have the exponent of p_1 set to its correct value, the exponents of each p_i for $i \geq 3$ set to e_i, and force the algorithm to search for the value of the exponent of p_2. Next comes a group of instances forcing the algorithm to search for the exponent of p_3, and so on. □

[3] To be precise, the adversary argument gives a stronger bound on $opt(\mathcal{L}^k(n))$ for all $n \geq 2^7 \cdot 3^3 \cdot 5 \cdot 7 \cdot 11 \cdots 31 \approx 8.9 \times 10^{13}$. After this point, $\max_{m \leq n} \sum_{i=1}^{s} \lfloor \log(e_i + 1) \rfloor$ is strictly greater than $\max\{r \mid 2 \cdot 3 \cdot 5 \cdots p_r \leq 2n\}$.

COROLLARY 4.3.

$$opt(\mathcal{L}^k(n)) \geq k \left(\max_{m \leq n} \sum_{i=1}^{s} \lfloor \log(e_i + 1) \rfloor \right)$$

where $m = \prod_{i=1}^{s} p_i^{e_i}$.

Proof. The method used by the adversary in the proof of Theorem 4.2 can be easily extended to the general case. There are k rounds; in the ith round the adversary gives instances with all components but the ith set to 0. The values of the ith component are chosen according to the adversary strategy in the proof of Theorem 4.2. □

5. Mistake bounds. This subsection calculates $M_{\text{CLOSURE}}(\mathcal{L}^k(n))$, the mistake bound of the closure algorithm, and considers how close it is to $opt(\mathcal{L}^k(n))$. We will bound $M_A(\mathcal{L}^k(n)) = M_{\text{CLOSURE}}(\mathcal{L}^k(n))$ by noticing that every time the algorithm makes a mistake, its new hypothesis is a strict superset of its old hypothesis. At the end of the section we analyze a modified halving algorithm for the special case when $k = 1$.

Before stating the main theorem, we recall some facts about lattices. The standard definition of lattices in $\Re^k$ insists that the lattice be generated by k linearly independent basis vectors. Our definition allows less than k basis vectors, thus our lattices need not have full "rank."

DEFINITION. The *rank* of lattice $\Lambda \subseteq \mathbf{Z}^k$ is the minimum of the ranks of all vector subspaces of $\Re^k$ that contain Λ.

Thus any lattice of rank r can be written as $\{\sum_{i=1}^{r} z_i x_i \mid z_i \in \mathbf{Z}\}$ for some r basis vectors $x_i \in \mathbf{Z}^k$.

It is easy to see that any integer lattice in $\mathbf{Z}^k$ with rank $r < k$ can be rotated into $\Re^r$ (i.e., the last $k - r$ coordinates of every point in the rotated lattice are zeros). For every set of r basis vectors that determine the same lattice, the volume of the r-dimensional parallelepiped that they generate is the same. Furthermore, rotating the lattice into $\Re^r$ preserves volume. The volume of the parallelepiped is called the *determinant* of the lattice [7]. We write $\det \Lambda$ to denote the determinant of lattice Λ. If $\Lambda \subseteq \mathbf{Z}^k$, then $\det \Lambda = 0$ only if Λ is O, the null lattice containing only the origin. Otherwise, $\det \Lambda$ is a positive integer.

THEOREM 5.1. *Let* $O = \Lambda_0 \subset \Lambda_1 \subset \Lambda_2 \subset \cdots \subset \Lambda_m$ *be a sequence of distinct lattices of* $\mathbf{Z}^k$ *where each* Λ_i, *for* $1 \leq i \leq m$, *is the closure of* Λ_{i-1} *plus some* $x_i \in (\mathbf{Z}^k \setminus \Lambda_{i-1})$ *and every component of every* x_i *has absolute value at most* n. *Then*

$$m \leq k + \left\lfloor k \log(n\sqrt{k}) \right\rfloor.$$

Proof. There are at most k values of i for which Λ_{i+1} can have greater rank than Λ_i.

Consider now the case where Λ_i and Λ_{i+1} have the same rank. Since Λ_i is a sublattice of Λ_{i+1}, we have $t \det \Lambda_{i+1} = \det \Lambda_i$ for some integer $t \geq 2$ [7].[4] Thus every time the rank of the lattice stays the same, we decrease the determinant by at least a factor of 2.

On the other hand, when the rank of Λ_{i+1} is greater than the rank of Λ_i, then the determinant can increase. The first lattice, Λ_1, is simply all integer multiples of some particular $x_1 \in \mathbf{Z}^k$, so its volume is $\|x_1\| \leq n\sqrt{k}$, where $\|x\|$ denotes the

[4] In geometry of numbers, t is called the index of Λ_i in Λ_{i+1}.

Euclidean norm of vector x. In general, when Λ_i has rank r and Λ_{i+1} has rank $r+1$, one set of basis vectors for Λ_{i+1} will be x_{i+1} together with the basis vectors of Λ_i. The $(r+1)$-dimensional volume of the parallelepiped associated with Λ_{i+1} is $\det \Lambda_i$ times the distance from x_{i+1} to the hyperplane containing Λ_i. Hence we have

$$\begin{aligned}\det \Lambda_{i+1} &\leq \|x_{i+1}\| \det \Lambda_i \\ &\leq n\sqrt{k} \det \Lambda_i.\end{aligned}$$

Thus $\det \Lambda_1 \leq n\sqrt{k}$, and the determinant is multiplied by no more than $n\sqrt{k}$ in at most $k-1$ other steps. In all other steps the determinant is divided by at least 2, and $\det \Lambda_m \geq 1$. Therefore $m \leq k + \lfloor k \log(n\sqrt{k}) \rfloor$. □

COROLLARY 5.2. $M_{\text{CLOSURE}}(\mathcal{L}^k(n)) \leq k + \lfloor k \log(n\sqrt{k}) \rfloor$.

Comparing Corollary 5.2 to Corollary 4.1 we see that the mistake bound achieved by the closure algorithm is indeed within a $\log\log n$ factor of optimal (assuming $n \gg k$).

By giving example sequences for which the closure algorithm makes the maximum number of mistakes we next show that for infinitely many choices of k and n, the bounds of Theorem 5.1 and Corollary 5.2 are tight. These sequences are constructed using Hadamard matrices. A *Hadamard matrix* is a square matrix where the entries are ± 1 and the columns are orthogonal to each other. Using n times the columns of a $k \times k$ Hadamard matrix, one can force the closure algorithm to make k mistakes while the volume of the closure algorithm's hypothesis grows to $(\sqrt{kn^2})^k$. Let n be a power of 2 and k a power of 4, so this volume is a power of 2. Consider now a $k \times k$ lower-triangular matrix whose columns form a basis for the hypothesized lattice. Since the determinant of this matrix is the product of the diagonal elements and equals the volume of the lattice, each of the diagonal elements is a power of 2. In particular, the element in the lower-right corner is some $\pm 2^l$. We can reduce this element by a factor of two (without changing the other diagonal elements) by giving the closure algorithm the positive example $(0, 0, \cdots, 0, 2^{l-1})$. After the corner element is reduced to ± 1, we can start reducing the diagonal element on the second-to-last row, and so on. The number of additional mistakes made by the closure algorithm is $\log(\sqrt{kn^2})^k = k \log(n\sqrt{k})$. Thus for infinitely many choices of k and n, the bound of Theorem 5.1 and Corollary 5.2 can be achieved.

Surprisingly, the mistake bound of the closure algorithm (together with Theorem 2.1) gives us our tightest upper bound on the VC dimension of $\mathcal{L}^k(n)$.

COROLLARY 5.3. $\text{VCdim}(\mathcal{L}^k(n)) \leq k + \lfloor k \log(n\sqrt{k}) \rfloor$.

For the case $k = 1$, when instances are integers and concepts are sets of multiples, instead of using Algorithm A, we can implement a modified version of the halving algorithm [5], [19], [4]. This modified halving algorithm has a basically optimal mistake bound in this case, but requires more computation than the closure algorithm. The modified halving algorithm predicts "−" on every instance except 0 until it makes a mistake on some instance m. The target concept is then known to be all multiples of one of the divisors of m. The modified halving algorithm predicts so that each time it makes a mistake, the number of possible target concepts is cut by at least half (i.e., it predicts as the majority of the remaining target concepts do). Note that the modified halving algorithm factors m, but that is one factorization for the whole run of the algorithm, rather than one per mistake.

THEOREM 5.4. *On instances of absolute value at most n, the modified halving algorithm achieves a mistake bound of* $1 + \max_{m \leq n} \lfloor \log \tau(m) \rfloor$ *where* $\tau(m)$ *is the number of positive divisors of* m.

Proof. The algorithm makes one mistake on the first positive example it sees. Call this example m. Without loss of generality, $0 \leq m \leq n$. Once m has been seen, the divisors of m are the only candidates for being the target concept, and the modified halving algorithm cuts the number of candidates by at least half with every mistake. Hence the mistake bound is, as claimed, $1 + \max_{m \leq n} \lfloor \log \tau(m) \rfloor$. □

Note that the bound of the theorem can be rewritten as

$$1 + \max_{m \leq n} \left\lfloor \sum_{i=1}^{s} \log(e_i + 1) \right\rfloor ,$$

where $m = \prod_{i=1}^{s} p_i^{e_i}$. Thus the mistake bound of the modified halving algorithm is just slightly above Theorem 4.2's lower bound of $\max_{m \leq n} \sum_{i=1}^{s} \lfloor \log(e_i + 1) \rfloor$.

6. Generalizations and applications of Algorithm A.

6.1. Arbitrary Euclidean domains. We need not limit ourselves to lattices of $\mathbf{Z}^k$. Algorithm A can in fact learn submodules of a free D-module for any Euclidean domain D. Careful examination of the algorithm shows that it does not rely on any properties of $\mathbf{Z}$ not possessed by all Euclidean domains. This generalization gives us learning algorithms for various exotic instance spaces such as k-tuples of Gaussian integers. Also, if we take D to be a field, then Algorithm A becomes Algorithm V for learning vector subspaces.

Unfortunately, however, the analysis of the mistake bound given in Theorem 5.1 no longer carries through. The difficulty is that we can no longer find a bound on any quantity analogous to the determinant of a lattice when we go from one module to a supermodule of higher rank.

6.2. Rational lattices. A slight modification of Algorithm A learns rational lattices (where the basis vectors consist of rationals rather than integers). After the derived examples are written with a common denominator, Algorithm A can be used whenever a mistake is made. By the argument of Theorem 5.1, the modified Algorithm A has a mistake bound of k plus the maximum number of times the determinant (volume) of the hypothesized lattice can be divided by an integer greater than 1. The determinant of the hypothesis is upper bounded by $(\sqrt{kn^2})^k$ where n is the largest component seen by the algorithm and lower bounded by v, the determinant of the target lattice. Therefore, the algorithm makes at most $k + \lfloor k \log(n\sqrt{k}) - \log v \rfloor$ mistakes. However, finding a common denominator and operating on the consequently larger numerators makes the modified algorithm computationally more expensive than Algorithm A applied to integer lattices.

6.3. Cosets of lattices and Algorithm A^+. A simple trick allows us to generalize the class of concepts we can learn from lattices to arbitrary cosets of lattices (viewing the lattice as an abelian group). The algorithm still responds "Negative" until it makes a mistake on some positive instance x. We now run the basic Algorithm A given in the Appendix, with the addition that x is subtracted from every instance. For the remainder of the paper, we use "A^+" to denote this modification of Algorithm A. Note that the mistake bound of Algorithm A^+ when learning cosets of $\mathcal{L}^k$ restricted to the domain $\{-n, \cdots, 0, \cdots, n\}^k$ is at most 1 plus the mistake bound of Algorithm A on $\mathcal{L}^k(2n)$ (the subtraction doubles the bounds on the size of the components). This leads to a mistake bound of

$$1 + k + \left\lfloor k \log(2n\sqrt{k}) \right\rfloor = 1 + 2k + \left\lfloor k \log(n\sqrt{k}) \right\rfloor$$

for Algorithm A^+ which is $k+1$ larger than Algorithm A's mistake bound when learning $\mathcal{L}^k(n)$.

6.4. Abelian groups. Consider the following on-line word problem for groups over a set of k generators:

> Given a sequence of words using the generators and their inverses as letters, predict for each word whether it is equal to some fixed element with respect to a hidden target group over the generators.

The goal is to minimize the number of mistakes.

In the case of abelian groups, words over the generators and their inverses can be represented as k-tuples. All words that are equal to some particular element with respect to a hidden abelian group form the coset of an integer lattice of $\mathbf{Z}^k$. Thus Algorithm A^+ leads to an efficient solution to this learning problem for abelian groups with a mistake bound of $1 + 2k + \lfloor k \log(n\sqrt{k}) \rfloor$, where n is a bound on the maximum word length of all instances seen. Since the word problem for general groups is undecidable, it is unlikely that there is an efficient learning algorithm for nonabelian groups. (See [10] for related work on permutation groups.)

6.5. Learning some commutative regular languages. We call a language L over alphabet Σ *commutative* if whenever some word w is in L, then all permutations of w are also in L. When learning such languages we can represent words as k-tuples of nonnegative integers, where $k = |\Sigma|$. Each component of the k-tuple equals the number of times the corresponding letter occurs. We use π to denote this mapping from words to tuples. If language L is commutative and $\pi(w) = \pi(w')$, then word $w' \in L$ if and only if $w \in L$. Therefore learning a commutative language is similar to learning a set of tuples that have nonnegative components.

For any subset S of $\mathbf{Z}^k$, we call the set of all tuples in S that have only nonnegative components the *nonnegative restriction* of S. If we have a class of languages where, for each language in the class, the image under π of the words in the language is the nonnegative restriction of a coset of a lattice, then Algorithm A^+ can be used to efficiently learn that class of languages. Simply use π to convert each word into a tuple and learn the tuples. The hypothesis of Algorithm A^+ may include tuples with negative components, however, no mistakes will be made on these tuples as they are not in the domain. The bulk of this subsection is devoted to showing that the images under π of a certain subclass of commutative regular languages (defined below) are the nonnegative restrictions of cosets of lattices, and thus can be efficiently learned by Algorithm $A^+$with a mistake bound of $1 + 2|\Sigma| + \lfloor |\Sigma| \log(n\sqrt{|\Sigma|}) \rfloor$.[5]

Note that many commutative nonregular languages can also be learned using Algorithm A. For example, the language over a, b containing all words with one more a than b is commutative but not regular, and its image under π is the coset of a lattice.

We now present several definitions concerning DFAs.

DEFINITION. A DFA is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$ where $q_0 \in Q$, $F \subset Q$, and δ is a partial function from $Q \times \Sigma$ to Q.[6] The elements of Q are called *states*; δ is called the *transition function*; q_0 is called the *start* state; and the states in F are called

[5] Abe has recently [2] discovered a learning algorithm (using different parameters) for arbitrary commutative regular languages. The mistake bound of his algorithm grows exponentially in $|\Sigma|$.

[6] We extend the transition function to words in the obvious way; if $a \in \Sigma$ and $w = aw'$, then $\delta(q, w) = \delta(\delta(q, a), w')$.

final states. The language *accepted* by this DFA is the set of all words w such that $\delta(q_0, w) \in F$.

DEFINITION. A DFA $M = (Q, \Sigma, \delta, q_0, F)$ is *closed* if for each final state $q_f \in F$ there is some word w_f such that $\delta(q_f, w_f) = q_0$.

DEFINITION. We say a DFA $M = (Q, \Sigma, \delta, q_0, F)$ is *canonical* if it has the following properties:

1. Every state can reach a final state.
2. If the language accepted by M is commutative, then we also require that for all states q and all words w and w', $\delta(q, ww') = \delta(q, w'w)$.

Given a closed DFA, one can create (by deleting and merging states) a closed canonical DFA that accepts the same language.

In the remainder of this section Σ denotes the set of useful letters—those on which the transition function is defined for some state, and k denotes the cardinality of Σ.

DEFINITION. A DFA $M = (Q, \Sigma, \delta, q_0, F)$ is *invertible*[7] if for each letter in Σ, every state in Q has both an incoming and an outgoing transition for that letter.

LEMMA 6.1. *Any closed canonical* DFA *accepting a commutative regular language is invertible.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a closed canonical DFA accepting a commutative language L. It suffices to show that every state in Q has at least one incoming transition for every letter of Σ. Since each state has at most one outgoing transition for every letter of Σ, this implies by the pigeonhole principle that for each state there is exactly one incoming and exactly one outgoing transition for every letter.

Let a be an arbitrary letter in Σ and q be an arbitrary state in Q. Since every state reaches a final state in F there is a word $u_1 a u_2$ that is accepted, i.e., $\delta(q_0, u_1 a u_2) = q_f \in F$. Let v be a word leading from the start state to q, and w be a word leading from q_f to the start state. Now $u_1 a u_2 w v$ leads to q and since $u_1 u_2 w v a$ is a permutation of the latter word it leads to q as well. Thus for the state $q' = \delta(q_0, u_1 u_2 w v)$, we have $\delta(q', a) = q$. □

Remark. Note that exactly one incoming and one outgoing transition per letter property does not hold if we remove the condition that the start state be reachable from the final states. For instance, the canonical DFA for $L_a = \{a\}$ has no outgoing transitions from its final state, even though L_a is commutative.

With an invertible DFA, one can talk about the inverses of letters. For the alphabet $\Sigma = \{\sigma_1, \cdots, \sigma_k\}$ we define the inverse alphabet Σ^{-1} to be the set of new symbols $\{\sigma_1^{-1}, \cdots, \sigma_k^{-1}\}$. The *extended alphabet* is $\Sigma \cup \Sigma^{-1}$.

DEFINITION. The *extension* of an invertible DFA $M = (Q, \Sigma, \delta, q_0, F)$ is the DFA $M' = (Q, \Sigma \cup \Sigma^{-1}, \delta', q_0, F)$, where for each $a \in \Sigma$ and $q \in Q$:

$$\delta'(q, a) = \delta(q, a),$$
$$\delta'(q, a^{-1}) = r \text{ such that } \delta(r, a) = q.$$

LEMMA 6.2. *For any closed canonical* DFA M *accepting a commutative language* L*, the language accepted by the extension of* M *is commutative.*

[7] Invertible DFAs with a single final state are *zero-reversible.* Angluin [3] presents an algorithm for learning the languages accepted by zero-reversible DFAs (as well as for the more general class of r-reversible DFAs). However, those algorithms learn in the limit and no bound was given on the number of mistakes made.

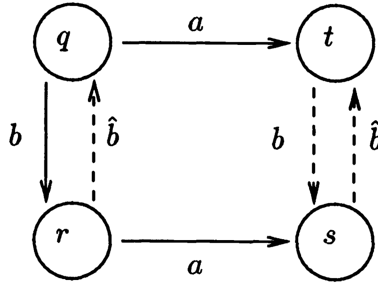


FIG. 2. *Detail of proof of Lemma 6.2: $\delta(q, ab) = \delta(q, ba)$. Dashed lines indicate the deduced transitions.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be such a DFA. By Lemma 6.1, each state has exactly one incoming and one outgoing transition for each letter in Σ. This means that M' is a DFA with transitions defined at each state for every letter in $\Sigma \cup \Sigma^{-1}$.

Now we argue that the language accepted by M' is commutative. It is sufficient to show that for all $q \in Q$, $\delta'(q, ab) = \delta'(q, ba)$ for arbitrary $a, b \in \Sigma \cup \Sigma^{-1}$. There are three cases:

1. Both $a, b \in \Sigma$. In this case, δ' is the same as δ.
2. Both $a, b \in \Sigma^{-1}$. Say $a = \hat{a}^{-1}$ and $b = \hat{b}^{-1}$, where $\hat{a}, \hat{b} \in \Sigma$. Now $\delta'(q, ab)$ must be some state r such that $\delta(r, \hat{b}\hat{a}) = q$. Since M is a canonical machine for a commutative language, $\delta(r, \hat{a}\hat{b}) = \delta(r, \hat{b}\hat{a}) = q$. Therefore $\delta'(q, ba) = r$ as well.
3. Let the letter $a \in \Sigma$, and the letter $b \in \Sigma^{-1}$. Again, say $b = \hat{b}^{-1}$ where $\hat{b} \in \Sigma$. By Lemma 6.1, every state has an entering and exiting transition for each letter in Σ, and hence for every letter in Σ^{-1} as well. Now let $r = \delta'(q, b)$, let $s = \delta'(r, a)$, and let $t = \delta'(q, a)$. $\delta'(q, ba)$ is obviously s. Now we need to calculate $\delta'(q, ab) = \delta'(t, b)$.
   By the commutativity of δ', $\delta(r, a\hat{b}) = \delta(r, \hat{b}a)$. Now since $r = \delta'(q, b)$, it follows that $\delta(r, \hat{b}) = q$. Therefore $t = \delta(r, \hat{b}a) = \delta(r, a\hat{b}) = \delta(s, \hat{b})$. Thus, as desired, $s = \delta'(t, b)$. (See Fig. 2.) □

Remark. Again, the property that the start state is reachable from the final states is crucial. Even if each state in a commutative canonical DFA has at most a single incoming transition, the extension of the DFA may not be commutative. Consider the two-state extended DFA for the commutative language $\{a\}$. The string $aa^{-1}a$ is accepted by that DFA, but the string $a^{-1}aa$ is not, because there is no transition out of the start state for a^{-1}.

When the extended DFA is commutative, it makes sense to talk about its behavior given simply letter counts from the extended alphabet, rather than specific words. We extend π as follows:

$$\pi(w) = ((\text{number of } \sigma_1 \in w) - (\text{number of } \sigma_1^{-1}), \cdots, \\ (\text{number of } \sigma_k \in w) - (\text{number of } \sigma_k^{-1} \in w)).$$

If a word w over the extended alphabet contains more occurrences of a letter σ_i^{-1} than σ_i, then the ith component of $\pi(w)$ is negative. Note that when learning a language L, neither the language nor the input include words with characters in Σ^{-1}. Although the hypothesis of Algorithm A^+ may contain these words, they are not in

the domain, and thus no mistakes will be made on them. The inverse alphabet is used only as a tool in the proofs.

We will show that closed canonical DFAs that accept a commutative language are essentially Cayley graphs [8], [11], [18].

DEFINITION. A directed multigraph (possibly with self-loops) where the edges are labeled with elements from an alphabet Σ is a *Cayley graph* if it has the following properties:

1. For each letter in Σ, every vertex has exactly one incoming and one outgoing edge labeled with that letter. This means that for each vertex, each word over the extended alphabet $\Sigma \cup \Sigma^{-1}$ describes an undirected path starting at that vertex (if the next letter in the word is some $\sigma \in \Sigma$ then follow the edge labeled with σ leaving the current vertex and if the next letter is some $\sigma^{-1} \in \Sigma^{-1}$ go to the tail of the edge labeled with σ entering the current vertex).
2. If a word over the extended alphabet describes a closed path starting at some vertex, then that word describes a closed path starting at every vertex in the graph.

DFAs naturally define a directed graph: The states are the vertices and the transitions correspond to directed edges.

LEMMA 6.3. *The directed graph defined by a closed canonical* DFA *accepting a commutative regular language is a Cayley graph.*

Proof. Let $M = (Q, \Sigma, \delta, q_0, F)$ be such a DFA and $M' = (Q, \Sigma \cup \Sigma^{-1}, \delta', q_0, F)$ be the extension of M. The first property follows from the fact that M is invertible (Lemma 6.1). For the proof of the second property, let state $q \in Q$ and word w over the extended alphabet be such that $\delta'(q, w) = q$. We need to show that for all states $r \in Q$, $\delta'(r, w) = r$ holds. By the definition of closed canonical DFAs there is a string $x \in \Sigma^*$, such that $\delta'(q, x) = r$. By Lemma 6.2, M' is commutative and thus $r = \delta'(q, x) = \delta'(q, wx) = \delta'(q, xw)$. We conclude that w also leads from r to r. □

Cayley graphs over the alphabet Σ correspond to groups over the generator set Σ as follows [18]: The vertices are the elements of the group and after fixing a start state (which becomes the neutral element) the words leading from the start state to a particular vertex are the words over the generators and their inverses that equal the group element corresponding to that vertex. Also, for each group generated by Σ there is a Cayley graph based on the above correspondence [18].

This implies that any closed canonical DFA (with alphabet Σ) accepting a commutative language corresponds to an abelian group generated by Σ. The language accepted by the DFA consists of all words over Σ that are equal to one of the group elements whose corresponding state is a final state. The abelian group defines a lattice and the language is the union of positive restrictions of cosets of that lattice, one coset for each final state. This proves the main theorem of this section.

THEOREM 6.4. *Let M be a* DFA *accepting a commutative language that is closed and has a single final state. Then the image under π of the language accepted by M is the positive restriction of a coset of a lattice.*

COROLLARY 6.5. *The class of commutative regular languages accepted by closed* DFA*s with one final state can be learned with a mistake bound of*

$$1 + 2|\Sigma| + \left\lfloor |\Sigma| \log(n\sqrt{|\Sigma|}) \right\rfloor,$$

where Σ is the alphabet and n is the length of the longest word seen.

Remark. A set of tuples in $\mathbf{Z}^k$ is the coset of a lattice if and only if whenever x, y, and z are in the set, then the tuple $x-y+z$ is also in the set. Unfortunately, it is not true that a subset of $\mathbf{N}^k$ is the nonnegative restriction of a lattice coset if and only if whenever x, y, and z are in the subset, then $x-y+z$ is either in the subset or has a negative component. Consider the set $\{(1,1,1),(3,0,0),(0,3,0)\}$. Every $x-y+z$ combination of these three vectors either has a negative component or is already in the set, but any lattice coset containing these three vectors also contains $(0,0,3)$. This is the simplest counterexample since for $k \leq 2$ the above characterization of nonnegative restrictions of cosets of lattices does hold.[8]

Corollary 6.5 is somewhat surprising in light of the results of Pitt and Warmuth [21]. They identify a small subclass $\mathcal{C}_k$ of the closed commutative regular languages over k letters, called counter languages, and show that for any $k \geq 2$ and any polynomial Q, the problem: "given a set of examples (from some $L \in \mathcal{C}_k$ accepted by a DFA of s states), find a DFA or NFA with fewer than $Q(s)$ states consistent with the examples" is NP-hard [21].[9] Algorithm A^+ bypasses that hardness result by representing its hypothesis as a coset of a submodule rather than as a DFA. The resulting algorithm for learning $\mathcal{C}_k$ makes at most $1+2k+\lfloor k\log(n\sqrt{k})\rfloor$ mistakes, where n is the length of the longest word seen.

We now define a number of subclasses of regular languages and give lower bounds on their VC dimensions (and hence the number of mistakes made on them by any learning algorithm). By this method we will show that the mistake bound of Corollary 6.5 is within a $\log\log n$ factor of optimal.

DEFINITION. Let $\mathrm{CCS}_{k,n}$ be the class of commutative regular languages over alphabets of size k accepted by closed DFAs having a single final state and restricted words of length at most n ($\mathrm{CCS}_{k,n}$ is the class of Corollary 6.5). Let $\mathrm{REG}_{k,n}$ and $\mathrm{CREG}_{k,n}$ be the class of all regular languages and all commutative regular languages, respectively, over alphabets of size k restricted to words of length at most n.

LEMMA 6.6.

1. $\mathrm{VCdim}(\mathrm{CCS}_{k,n}) \leq 1+k+\lfloor k\log(n\sqrt{k})\rfloor$ *and for every* $\epsilon > 0$ *and for all sufficiently large* n, $\mathrm{VCdim}(\mathrm{CCS}_{k,n}) \geq k(1-\epsilon)\ln n/\ln\ln n$.
2. $\mathrm{VCdim}(\mathrm{REG}_{k,n}) = \frac{k^{n+1}-1}{k-1}$ *if* $k>1$ *and* $n+1$ *if* $k=1$.
3. $\mathrm{VCdim}(\mathrm{CREG}_{k,n}) = \binom{n+k}{k}$.

Proof. The upper bound of part 1 of Lemma 6.6 follows from the mistake bound of the algorithm for learning $\mathrm{CCS}_{k,n}$ given in Corollary 6.5. For the lower bound first observe that the commutative languages over the single letter σ accepted by closed DFAs with a single final state are all languages of the form $\sigma^i(\sigma^j)^*$. Thus the letter counts of these languages are the positive restrictions of shifted one-dimensional lattices. By Corollary 3.2, one-dimensional lattices (ignoring possible shifts) restricted to $\{-n,\cdots,0,\cdots,n\}$ have VC dimension larger than $(1-\epsilon)\ln n/\ln\ln n$, for every $\epsilon>0$ and for all sufficiently large n. The shattered set used to prove the lower bound for one-dimensional lattices (proof of Theorem 3.1) consisted only of positive numbers. Thus the same lower bound applies for $\mathrm{CCS}_{1,n}$.

Positive restrictions of one-dimensional lattices correspond to languages of the

[8] This example corresponds to the language L containing *aaa*, *bbb*, and all permutations of *abc*. Language L is commutative and zero-reversible [3] and thus Algorithm A^+ is unable to learn all of the commutative zero-reversible languages over three or more letters.

[9] If k is an input to the problem then it is even NP-hard to produce a consistent NFA of superpolynomial size: for any $0<\epsilon<1$ it is NP-hard to find a consistent NFA of size $s^{(1-\epsilon)\log\log s}$ [22].

form $(\sigma^j)^*$, i.e., the canonical DFAs accepting these languages have a single final state that equals the start state. Let C denote the subclass of these languages over one letter. To complete the proof of part 1 it suffices to show that $\text{VCdim}(\text{CCS}_{k,n}) \geq k\text{VCdim}(C)$. The proof of this is similar to the proof of Theorem 3.3.

Let S_1 be a set of words over the same letter shattered by C. Let $S = \bigcup_{i=1}^{k} S_{\sigma_i}$, where S_{σ_i} is a copy of S_1 using σ_i as the single letter. Clearly, $|S| = k\,|S_1|$. To show that S is shattered by $\text{CCS}_{k,n}$, let T be an arbitrary subset of S and $T_i = T \cap \sigma_i{}^*$. Now for each T_i there is a DFA M_i over the letter σ_i accepting T_i. Using a standard cross product construction it is easy to build from $M_1, \cdots, M_k$ a commutative DFA over the alphabet $\sigma_1, \cdots, \sigma_k$ accepting T. Since for each M_i the start state equals the single final state, the same holds for the new DFA. Thus this DFA witnesses the fact that T is in $\text{CCS}_{k,n}$.

For the proof of part 2 of Lemma 6.6, observe that the class $\text{REG}_{k,n}$ shatters its entire domain of all words of length at most n, since all subsets of the domain are in the class. The size of the domain is $\sum_{i=0}^{n} k^i$.

In the commutative case (part 3), the class $\text{CREG}_{k,n}$ also shatters its entire domain, which can be characterized by the set of all k-tuples of nonnegative integers whose components sum to at most n. The size of this domain is $\binom{n+k}{k}$. □

6.6. Discussion. Part 1 of Lemma 6.6 shows that the mistake bound of Algorithm $A^+$when used to learn $\text{CCS}_{k,n}$ is within a $\log\log n$ factor of optimal and parts 2 and 3 indicate that it is much harder to learn arbitrary regular languages or even arbitrary commutative regular languages.

7. Nested differences. Using the results obtained in a companion paper [14], we can apply Algorithm A in the construction of a number of master algorithms that learn nested differences of lattices. Let $\text{DIFF}(\mathcal{L}^k)$ be the class of concepts of the form $\Lambda_1 - (\Lambda_2 - (\Lambda_3 - \cdots - (\Lambda_{p-1} - \Lambda_p))\cdots)$, where each $\Lambda_i \in \mathcal{L}^k$. Thus each concept in $\text{DIFF}(\mathcal{L}^k)$ is a nested difference of lattices. We call p the *depth* of the concept. The master algorithms learn the class $\text{DIFF}(\mathcal{L}^k)$ with a mistake bound that is p times the bound for single lattices. The master algorithms can be used to learn nested differences of any intersection-closed class.[10]

In this section we sketch only a single master algorithm for learning $\text{DIFF}(C)$, where C is any intersection-closed class, and discuss how it can be adapted to learn nested differences of those concept classes where Algorithm A (or its coset modification) was applied. Thus this modified master algorithm can be used to learn nested differences of cosets of lattices, nonhomogeneous vector spaces, commutative regular languages accepted by DFAs whose single final state reaches the start state, etc. The mistake bound, the efficiency, and (generally) the VC dimension grow linearly with the depth of the nested difference. Thus the master algorithm efficiently learns these classes with good mistake bounds.

The master algorithm[11] for $\text{DIFF}(C)$ keeps track of a finite sequence of closures. The depth of the master algorithms hypothesis is the number of closures in the sequence. In [14], each closure is represented by a minimal set of instances that defines the closure. When given a new instance x, the master predicts on x by the following rule. Let the lth closure be the first one that does not contain x. (If x is in all of

[10] In [14] we also give master algorithms for the case when each concept Λ_i is in the union of several concept classes, each of which is intersection-closed, and for the case where the innermost concept Λ_p is from a concept class that is not necessarily intersection-closed.

[11] This is the "space efficient master algorithm" of [14] for learning $\text{DIFF}(C)$.

the closures, then let l be 1 + the depth of the hypothesis.) If l is even, then predict "+" and if l is odd, predict "−." When a mistake is made, x is added to the lth closure. If l is one larger than the depth of the current hypothesis, then the depth of the hypothesis is increased by initializing the lth closure to the closure of $\{x\}$.

The above algorithm applies to learning nested differences of lattices, since they are intersection-closed. A different copy of Algorithm A can be used to efficiently compute each closure of the sequence. It is shown in [14] that the mistake bound of the master algorithm for learning concepts in DIFF(C) of depth p is at most p times the mistake bound of the closure algorithm applied to C.

In §6.3 we gave a simple trick for learning cosets of lattices. A slight modification of the master algorithm lets this trick be used at each position in the sequence. Let x_i be the first mistake made at each position i in the sequence of closures. Example x_i is not directly used to form the ith closure, but rather is remembered as the *shift* at position i. When predicting on a new instance x, l is now the first closure that does not contain $x - x_l$, the appropriately shifted example. (If $x - x_i$ is in the ith closure for all i between 1 and the depth of the hypothesis, then set l to 1 + the depth of the hypothesis.) When a mistake is made, the lth closure is adjusted to include $x - x_l$. If l is one larger than the depth of the hypothesis, then x_l is set to x, i.e., x becomes the shift of the new lth level and the depth of the hypothesis is increased.

This modified master algorithm learns nested differences of cosets of lattices, abelian groups, and commutative regular languages accepted by DFAs whose single final state reaches the start state. Its mistake bound is at most p times the mistake bound for cosets of lattices (which is $1 + k$ larger than the bound for lattices), where p is the depth of the target.

8. Conclusions. This paper contains a nontrivial algorithm that efficiently learns the basic combinatorial class of integer lattices. The algorithm leads to efficient learning algorithms for a large number of other classes. The mistake bounds of this algorithm, and most of its applications presented, are provably within roughly a $\log \log n$ factor of general lower bounds derived from the VC dimension.

9. Appendix: An implementation of the closure algorithm. This appendix gives a precise description of Algorithm A for on-line learning of the concept class $\mathcal{L}^k$ of integer lattices, and analyzes its running time. This algorithm is an implementation of the closure algorithm and was derived from the algorithm of Kannan and Bachem [15] for putting a matrix in Hermite normal form (HNF). Basically, we keep track of a basis for the smallest lattice containing all of the positive instances seen so far. Whenever a mistake is made, a new basis for the smallest lattice containing both the old lattice and the example on which the mistake was made must be found. Since Algorithm A is an implementation of the closure algorithm, it never makes mistakes on negative examples.

We first present the on-line Algorithm A and show that it computes the appropriate basis. In order to bound the size of the entries stored by Algorithm A, we present a batch algorithm A', which gets all of the examples at once. By an analysis similar to that used for HNF algorithms [25], [15], we bound the size of entries stored by A'. Finally, we argue that the values stored by Algorithm A after it has made its ith mistake are a subset of the values stored by Algorithm A'. Therefore the bound on the entries stored by Algorithm A' carries over to Algorithm A.

Algorithm A keeps a k by k lower triangular matrix M whose column span represents the current hypothesis. Matrix M is initially all 0, and gradually has nonzero columns added to it as positive examples are seen. Algorithm A will occasionally

exchange rows in M. This operation corresponds to changing the order of the components in examples. At any point, the permutation π reflects the row exchanges made by A, and the necessary adjustment to the components of new examples. The algorithm's current hypothesis is all (permuted) examples in CLOSURE(M), the (integer) column space of M. Algorithm A makes a mistake only when it gets a new positive example x where $\pi(x)$ is not in the column space of M. When this happens, Algorithm A updates M (and possibly π) so that the column space contains $\pi(x)$ in addition to the (possibly permuted) column space of the original matrix.

ALGORITHM A. Matrix M is initially the $k \times k$ matrix of 0's. Permutation π is initialized to the identity permutation on k elements. The variable z is initialized to 1 and contains the index of the leftmost all-zero column in M. Throughout, m_{ij} denotes the entry in the ith row and jth column of M. The following procedure is executed for each instance x.

1. Permutation:
   $x := \pi(x)$. The component ordering of x now agrees with the row ordering in M.
2. Prediction:
   Determine whether x can be written as an integer combination of the columns in the matrix. (Since M is lower triangular, this can be done by back-substitution with $O(k^2)$ arithmetic operations.) If so, predict "Positive," since it is certain that x is in the target lattice. Otherwise, predict "Negative." (If M is the zero matrix, then a positive prediction is made only on the zero vector.)
3. Update:
   If a mistake is made, then x replaces the all-zero column z in M.[12]
   To return M to normal form, perform the following operation, rather similar to Gaussian elimination:
   (a) For $i := 1$ to $z - 1$ do: if m_{iz} is not already 0, force it to 0 by the following:
      i. Use the extended GCD algorithm to find a, b, and g such that $am_{ii} + bm_{iz} = g = \gcd(m_{ii}, m_{iz})$.
      ii. Simultaneously update columns i and z of M. Replace column i with a times column i plus b times column z (and thus m_{ii} becomes g) and replace column z of M with m_{ii}/g times column z minus m_{iz}/g times the old value of column i. This "zeros out" the entry m_{iz}.
   (b) If column z now contains a nonzero entry then x is not linearly dependent on the previous examples. Let j be the first row containing a nonzero entry in column z. If this nonzero entry is negative, multiply column z by -1. Swap rows j and z in M, and swap the zth and jth elements in π. Finally, set z to $z + 1$, as the number of nonzero columns in M has increased.
   (c) Call REDUCE($z-1$, M) to ensure that each element to the left of a nonzero diagonal element is less than that diagonal element.
4. Get a new example and go to step 1.

Procedure REDUCE(i, M) [25]. This procedure performs elementary column operations to ensure that each below-diagonal element in both the first i columns and the first i rows is nonnegative and smaller than the (positive) diagonal element on the same row. The off-diagonal elements are examined from column $i - 1$ down to column 1, and from row $c + 1$ to row i within each column c.

```
for c := i − 1 down to 1 do
        for r := c + 1 to i do
                if (m_rc < 0) or (m_rc ≥ m_rr) then
                        subtract ⌊m_rc/m_rr⌋ times column r from column c
                        making 0 ≤ m_rc < m_rr
                end if;
        end for r;
end for c;
end REDUCE
```

[12] If M already contains k nonzero columns, then a temporary column $z = k+1$ is created to hold x. The temporary column is deleted after it is "zeroed out" by the following operations.

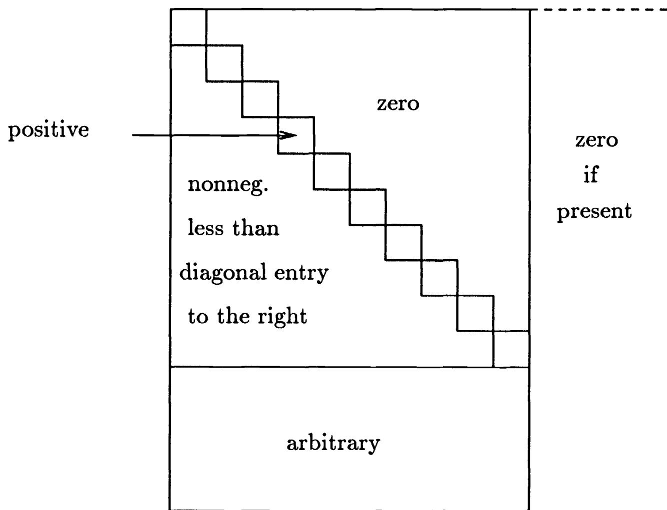


FIG. 3. *Pseudo-Hermite normal form.*

Throughout, M is kept in a pseudo-Hermite normal form (pseudo-HNF).[13]

DEFINITION. A matrix M_1 is a *pseudo-Hermite normal form* (see Fig. 3) of matrix M_2 if both:

- $M_1 = PM_2U$ where P is a permutation matrix and U is unimodular. Thus M_1 can be derived from M_2 by a series of row permutations and elementary column operations [15].
- M_1 is a lower-triangular matrix where in every column with any nonzero entry, the diagonal element is positive and each element to its left is nonnegative and less than that diagonal element.

Although these properties are not needed until we prove that the entries of M remain small, they are the motivation behind several of the operations in Algorithm A.

The strange order used in the REDUCE procedure (see Fig. 4) ensures that only previously reduced elements are added to (or subtracted from) the elements which have not yet been reduced (see [25]).

LEMMA 9.1. *Algorithm A correctly implements the closure algorithm.*

Proof. The row exchange in step 3(b) is reflected by an update to permutation π, which is applied to all future examples. Thus it suffices to show that whenever a prediction mistake is made, matrix M is updated so that its column span (ignoring the row permutation) includes x and no points not in the column span of $M \cup \{x\}$. This occurs when x is inserted into M at the beginning of the update step. It remains to show that the other operations on M do not change its (permuted) column span.

The operations in procedure REDUCE consist of adding a multiple of one column to another, and thus do not change the column span of M. Similarly, the multiplication of a column by -1 in step 3(b) does not change the column span. Finally, we

[13] The notion of pseudo-Hermite normal form is a generalization of Hermite normal form defined for nonsingular (integer) matrices.

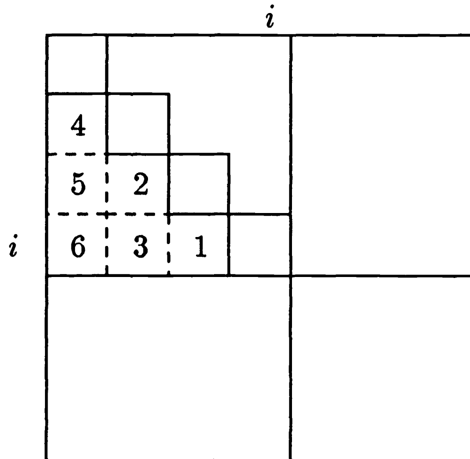


FIG. 4. *Illustration of the REDUCE procedure.*

show that the simultaneous update in step 3(a)ii does not change the column span of M.

It is easy to see that the new columns are in the lattice generated by the old columns. Furthermore, the old column z is a times the new column z plus m_{rz}/g times the new column r, and the old column r is $-b$ times the new column z plus m_{rr}/g times the new column r. Therefore, the lattice generated by M is not changed by the simultaneous updates. □

Although we now know that Algorithm A is correct, we have yet to show that it is efficient. The time taken by the GCD computations performed by Algorithm A, as well as the amount of space used by Algorithm A, depends on the size of the numbers stored in the $k \times k$ matrix M. To bound these entries we study a slightly different algorithm, A'. Algorithm A' uses unimodular column operations and row swaps to convert a $t \times t$ nonsingular matrix of padded examples, M', into pseudo-Hermite normal form. Using the techniques in [25], [15], we bound the entries of this matrix after each column is processed. We will also show that every nonzero entry stored in matrix M by Algorithm A is stored in M' by Algorithm A'.

To create the matrix M', fix the sequence of examples on which the closure algorithm makes t prediction mistakes and let e_i be the example on which the closure algorithm makes its ith mistake. (Thus e_1 will be the first nonzero example.) We assume that $t \geq k$ and that the e_i's have full row rank,[14] and pad the examples out to length t as follows. First, for $1 \leq i \leq t$, define $r(i)$ to be the row rank of the $k \times i$ matrix with columns $e_1, \cdots, e_i$. Note that $i - r(i)$ is the number of linearly dependent columns in this matrix and is a nondecreasing function of i. We now extend each column e_i to an e'_i of length t as follows. If $i = 1$ or $r(i) > r(i-1)$, then vector e'_i is e_i followed by $t - k$ "0"s. If $r(i) = r(i-1)$, then e'_i is e_i followed by $i - 1 - r(i)$ "0"s, a single "1," and another $t - k - i + r(i)$ "0"s. The ith column in matrix M' is e'_i, for $1 \leq i \leq t$. Since the e'_i's are linearly independent and of length t, matrix M', consisting of $e'_1, \cdots, e'_t$, is square and nonsingular. The last $t - k$ rows of M' are the *padding rows*, and any column with a nonzero entry in a padding row is a *padded column* (see Fig. 5).

[14] Additional examples where the closure algorithm would make mistakes can be appended to $\{e_1, \cdots, e_t\}$, making the assumptions true.

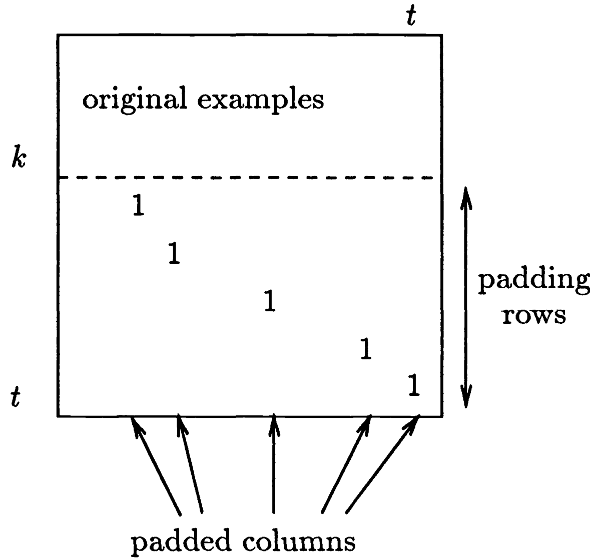


FIG. 5. *Initial structure of M'.*

Like the Hermite normal form (HNF) algorithm of Kannan and Bachem [15], our algorithm iteratively processes the columns from left to right, placing the principal minors of M' into (pseudo-) Hermite normal form.

That algorithm first preconditions the matrix using column permutations so that all of the principle minors in the resulting matrix are nonsingular. In our application we are given one example (column) at a time, so we replace the preconditioning step by on-the-fly row swaps. These row swaps will ensure that after processing column i, the $i \times i$ principal minor is nonsingular. Furthermore, we segregate the padding rows and padded columns from the normal rows and columns. Whenever a $c \times c$ principal minor has been placed into pseudo-HNF, there will be some number, say c', of processed nonpadded columns. These nonpadded columns will be in columns one through c', and the processed padded columns will be in columns $c' + 1$ through c. Similarly, rows one through c' will be nonpadding rows and rows $c' + 1$ through c will be padding rows (see Fig. 6).

To maintain this organization, at each iteration the new column is moved left to between columns c' and $c' + 1$. Similarly, some row below row $c - 1$ will be moved up between rows c' and $c' + 1$.

ALGORITHM A'. *This algorithm modifies matrix M', placing it into pseudo-*HNF.

- Initialize c' to 0. Variable c' counts the number of nonpadded columns which have been processed.
- For $c := 1$ to t put the $c \times c$ minor into pseudo-HNF as follows:
  1. For $i := 1$ to c' do: if m'_{ic} is not already 0, force it to 0 by the following:
     (a) Use the extended GCD algorithm to find a, b, and g such that $am'_{ii} + bm'_{ic} = g = \gcd(m'_{ii}, m'_{ic})$.
     (b) *Make m'_{ic} zero:* Simultaneously update columns i and c of M'. Replace column i with a times column i plus b times column c (and thus m'_{ii} becomes g) and replace column c of M to m'_{ii}/g times column c minus m'_{ic}/g times the old value of column i. This "zeros out" the entry m'_{ic}.
  2. If $m'_{cc} = 0$ then permute rows to make it nonzero. Let m'_{rc} be the topmost nonzero entry in column c below row c. Move row r to row c, shifting each row from row c through row $r - 1$ down one position. The column c is a padded column exactly

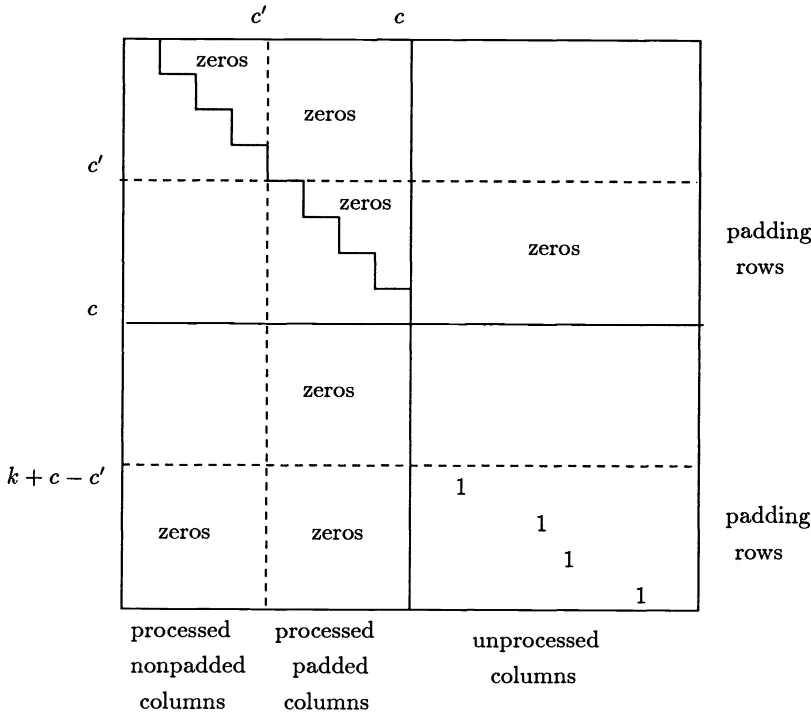


FIG. 6. *Structure of M' after c iterations.*

when $r \geq k + c - c'$.
3. If $m'_{cc} < 0$ then multiply column c by -1.
4. Permute the rows and columns to segregate the padding rows and padded column. (See Fig. 7.)
   (a) Move column c to column $c'+1$, shifting columns $c'+1$ through $c-1$ one column to the right.
   (b) Move row c to row $c'+1$ shifting rows $c'+1$ through $c-1$ one row down.
5. If column c was a nonpadded column then $c' := c' + 1$.
6. Call REDUCE(c, M') to ensure that the $c \times c$ principal minor of M' is in pseudo-HNF.

Note that Algorithm A' uses only columns 1 through c while processing the first c columns and placing the $c \times c$ principal minor into pseudo-HNF. Although the simultaneous transformations made in step 1(b) can affect the values of padding rows, we adopt the convention that this step never changes the padded/nonpadded status of a column. Also, the padded/nonpadded status of columns and the padding/nonpadding status of rows is carried along with them when rows and columns are permuted.

The simultaneous transformations of Step 1(b) are unimodular since the matrix

$$\begin{bmatrix} a & -m'_{rc}/g \\ b & m'_{rr}/g \end{bmatrix}$$

has determinant $am'_{rr}/g + bm'_{rc}/g = 1$. Note also that this transformation is identical to the one used in Algorithm A.

In some sense, the padded columns are used in only one iteration of the "for c"

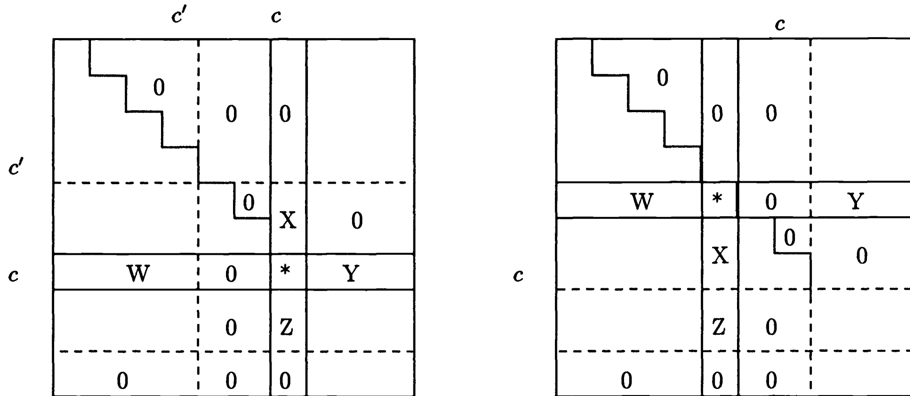


FIG. 7. *Illustration of step* 4.

loop of Algorithm A'. A padded column is created from an example that is a linear combination of the previous examples. Therefore, immediately after a padded column has been processed its only nonzero entries will be in padding rows.

It is not obvious that a processed padded column will always have zero entries in nonpadding rows. The padded column might be spoiled during step 1 while processing a future column, or during the REDUCE procedure if a nonpadded column is used to reduce one of its entries.

LEMMA 9.2. *A processed padded column always has a* 0 *in every nonpadding row.*

Proof. The proof is by induction on the number of processed padded columns. Let column c be the first padded column in the initial matrix M'. The lemma holds trivially until iteration c, when this padded column is processed. Column c was created from example e_c, which can be expressed as a linear combination of the previous examples. Columns 1 through $c-1$ represent a basis for the vector space generated by the previous examples, so all entries in column c on normal rows will be "zeroed out" during step 1. Thereafter, the row and column permutations guarantee that column c remains the last column in the pseudo-HNF principal minor. The only further references to column c are shifting it right a column, permuting its entries in steps 2 and 4(b), and in the procedure REDUCE where it is subtracted from other columns. None of these operations can create a nonzero entry in a nonpadding row.

Let there be $c-c' > 0$ processed padded columns before iteration c where column c is a padded column. The nonpadding rows of the first c' columns represent a basis for the vector space generated by the first c examples. During step 1, all entries in column c representing components of e_c will be zeroed out. The only other (nonpermutation) modifications to the column occur during the REDUCE procedure, where multiples of other padded columns may be subtracted from it. However, by the inductive hypothesis, the nonzero entries of these other columns occur only in padded rows. Therefore, the column's entries on nonpadding rows remain 0. □

COROLLARY 9.3. *Entries in a nonpadding row are not affected by adding multiples of padded columns during the* REDUCE *procedure.*

LEMMA 9.4. *At the end of iteration c, the first c columns of* M' *are converted to*

pseudo-HNF *form.*

Proof. Since the algorithm performs only row permutations and unimodular column operations, the first condition for a pseudo-HNF form is met. The algorithm ensures that the diagonal elements are positive (step 3), and the REDUCE procedure ensures that the entries to the left of the diagonal have the proper values. The remainder follows by induction on c.

If the newly processed column is a nonpadded column then, after step one, the first $c-1$ entries of c are 0. Each padding row is either in its original position or is above row c, so the first nonzero entry in column c is in a nonpadding row. Thus after step 2, row c will be a nonpadding row. By Lemma 9.2 that row contains a 0 in each processed padded column. Therefore, if the first $c-1$ columns were lower-triangular before step 4, then the first c columns are lower-triangular after that step.

If the newly processed column is a padded column, then at step 2 the only nonzero element in column c below row c is some multiple[15] of the original "1" in the padded example for that column. Before iteration c, this column was the only column containing a nonzero entry on that padding row, which becomes row c after step 2. Thus, the previously processed padded columns still have zero entries in the (new) row c, and if the first $c-1$ columns were lower-triangular before step 4, then the first c columns are lower-triangular after that step. □

LEMMA 9.5. *Let M^i be the modified matrix M' after i iterations of the main loop of Algorithm A'. The entries of M^i are no larger than $tn^{k+1}k^{k/2}$ for $1 \leq i \leq t$.*

Proof. Algorithm A' ensures that, at the end of iteration i, the $i \times i$ principal minor of M^i is a nonsingular, lower-triangular matrix with each entry between 0 and the value of the diagonal element to its right. Furthermore, each M^i is formed from M' by a series of row permutations and unimodular column operations, thus $M^i = P^i M' U^i$. Since the column operations involve only the first i columns,

$$U^i = \begin{bmatrix} U_i^i & 0 \\ 0 & I \end{bmatrix},$$

where U_i^i is an $i \times i$ unimodular matrix. Let M_i^i and M_i' denote the $i \times i$ principal minors of M^i and $P^i M'$, respectively. Thus, $M_i^i = M_i' U_i$ and[16]

$$U_i = (M_i')^{-1} M_i^i = \frac{\text{adj } M_i'}{\det M_i'} M_i^i.$$

Let $\text{adjmax}(M_i')$ be the largest absolute value of any entry in adj M_i'. Now when u_{jk} and m_{jk}^i are the entries in row j and column k of U_i and M_i^i, respectively,

$$|u_{jk}| \leq \sum_{l=1}^{i} |m_{lk}^i| \frac{\text{adjmax}(M_i')}{|\det M_i'|}.$$

The sum of the entries in any column contains at most one diagonal element. Since M_i^i is in Hermite normal form, every entry is nonnegative and each nondiagonal entry is less than the diagonal entry to its right. Therefore,

$$\sum_{l=1}^{i} |m_{lk}^i| \leq (m_{11}^i - 1) + (m_{22}^i - 1) + \cdots + (m_{ii}^i - 1) + 1$$

---

[15] In step 1 column c is repeatedly replaced by m'_{ii}/g times column c plus a multiple of some already processed column.

[16] adj M represents the adjoint of M.

$$\leq \prod_{l=1}^{i} m_{ll}^{i} = \det M^i = \det(M_i' U_i) = |\det M_i'|,$$

and so for each u_{jk},

$$|u_{jk}| \leq \operatorname{adjmax}(M_i').$$

Every element of adj M_i' is the determinant of an $(i-1) \times (i-1)$ minor of M_i'. Recall that the last $t-k$ rows of M' each contain a single 1 and $t-1$ 0's. Therefore, by cofactor expansion, the determinant of any large submatrix of M' is at most the determinant of a $k \times k$ matrix whose columns are in $\{e_1, \cdots, e_t\}$. Note that each entry in this smaller matrix is at most n, so its determinant contains $k!$ terms each at most n^k. Applying Hadamard's inequality (see, for example, [12]) shows that the determinant of any $k \times k$ matrix with entries bounded by n, and thus each u_{jk}, has absolute value at most $k^{k/2} n^k$.

As $M^i = P^i M' U^i$, each entry of M' has absolute value at most n, and each entry of U has absolute value at most $k^{k/2} n^k$; the absolute value of each entry of M^i is at most $tnk^{k/2} n^k$. □

In particular, $t \leq k + k\log(n\sqrt{k})$, the mistake bound of the closure algorithm. This gives us the following corollary.

COROLLARY 9.6. *After each iteration of the "for c" loop of Algorithm A', the largest entry in M^c is at most $k^{k/2} n^{k+1}(k + k\log(n\sqrt{k}))$, which can be written in $O(k\log(nk))$ bits.*

Using the above, we bound the size of numbers *during* iterations of Algorithm A'.

LEMMA 9.7. *During the computation of Algorithm A', each entry of M' requires at most $O\left(k\log^2(nk)\right)$ bits.*

Proof. The sizes of entries are changed only in the REDUCE procedure and step 1. Let $m = k^{k/2} n^{k+1}(k + k\log(n\sqrt{k}))$ be a bound on the largest absolute value of an entry in M' at the beginning of step 1. Each iteration of this step increases the entries in column c by a factor of at most $2m$, as both a and b found in Step 1(a) are at most m [15]. Since there are at most k iterations, the largest entry in column c at the end of step 1 is at most $m(2m)^k$. The entries in the other columns are bounded by the same expression.

The REDUCE procedure can also produce large intermediate results. Consider what happens as we reduce some column. The entries in the previously reduced columns are bounded by m. Each time an entry in the column is reduced, the unreduced entries in that column are increased by at most m times the value of the entry being reduced (reduced entries are never greater than m). Since there are t entries in each column that are originally bounded by $m(2m)^k$, the entries of the column never get larger than $m(2m)^k(1+m)^t$.

The maximum number of bits needed to represent an entry is roughly $t\log(m+1)+k+(k+1)\log m$. Plugging in the bounds on m and t gives us that $O\left(k^2\log^2(nk)\right)$ bits suffice to represent each entry (note that the hidden constants are small). □

Note that one of the key contributions of [25] is the clever order used by the REDUCE procedure. This lets them show that the maximum entry size *during* an iteration of their HNF algorithm is within a constant factor of the between-iteration bound. It appears likely that their techniques could achieve better bounds on the maximum entry size for Algorithm A' than the simple ideas used in the proof of Lemma 9.7.

We now formally state the relationship between Algorithm A and Algorithm A'.

LEMMA 9.8. *After Algorithm A has processed the cth mistake and Algorithm A' has processed the first c columns of M', each nonzero entry in M also appears in M'.*

Proof. It is easy to show by induction on c that the first c columns of M' are identical to the nonzero columns in M after the padded columns and padding rows are deleted. □

This allows us to apply Corollary 9.6 and Lemma 9.7 to Algorithm A.

THEOREM 9.9. *Using the uniform*[17] *cost measure, Algorithm A requires time $O\left(k^2\right)$ to make a prediction, and time $O\left(k^3 + k\log(nk)\right)$ to perform an update, where n is the largest absolute value of any component of any instance seen.*

Proof. Prediction time: The prediction time is just the time for back substitution, $O\left(k^2\right)$.

Update time: In step 3(a), updating matrix M can in general require k extended GCD operations to be performed, and the running time for extended GCD is proportional to the logarithm of the smaller of the two numbers. In our case, one of the two numbers will always be a diagonal element stored between iterations. By Lemma 9.5, the saved diagonal elements are at most $(k + k\log(n\sqrt{k}))k^{k/2}n^{k+1}$ and each extended GCD computation can be done in $O\left(k\log(kn)\right)$ time.

Finally, each iteration of the nested for loop of procedure REDUCE requires $O(k)$ subtractions, so this step takes time $O(k^3)$ altogether. □

Acknowledgment. We would like to thank David Haussler for insightful discussions.

REFERENCES

[1] N. ABE, *Polynomial learnability of semilinear sets*, in Second Workshop on Computational Learning Theory, Santa Cruz, CA, July 1989, Morgan Kaufmann, Los Altos, CA, pp. 24–40.

[2] ———, *Learning commutative deterministic finite state automata in polynomial time*, New Generation Computing, 8 (1991), pp. 319–336.

[3] D. ANGLUIN, *Inference of reversible languages*, J. Assoc. Comput. Mach., 29 (1982), pp. 741–765.

[4] ———, *Queries and concept learning*, Machine Learning, 2 (1987), pp. 319–342.

[5] J. M. BARZDIN AND R. V. FREIVALD, *On the prediction of general recursive functions*, Soviet Math. Dokl., 13 (1972), pp. 1224–1228.

[6] S. BOUCHERON, *Learnability from positive examples in the Valiant framework*, unpublished manuscript, 1988.

[7] J. W. S. CASSELS, *An Introduction to the Geometry of Numbers*, Springer-Verlag, Berlin, New York, 1959.

[8] H. S. M. COXETER AND W. O. J. MOSER, *Generators and Relations for Discrete Groups*, Third Edition, Springer-Verlag, New York, 1972.

[9] R. M. DUDLEY, *A course on empirical processes*, in Lecture Notes in Mathematics No. 1097, Springer-Verlag, Berlin, New York, 1984.

[10] A. FIAT, S. MOSES, A. SHAMIR, I. SHIMSHONI, AND G. TARDOS, *Planning and learning in permutation groups*, in 30th Annual IEEE Symposium on Foundations of Computer Science, 1989, pp. 274–279.

[11] I. GROSSMAN AND W. MAGNUS, *Groups and Their Graphs*, Vol. 14, New Mathematical Library, Mathematical Association of America, Washington, 1964.

[12] G. H. HARDY, J. E. LITTLEWOOD, AND G. PÓLYA, *Inequalities*, Second Edition, Cambridge University Press, Cambridge, U.K., 1952.

[13] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers*, Fourth Edition, Oxford University Press, Oxford, U.K., 1960.

[14] D. HELMBOLD, R. SLOAN, AND M. K. WARMUTH, *Learning nested differences of intersection-closed concept classes*, Machine Learning, 5 (1990), pp. 165–196.

[17] This is reasonable since we have just shown that the numbers involved remain reasonably small.

[15] R. KANNAN AND A. BACHEM, *Polynomial algorithms for computing the Smith and Hermite normal forms of an integer matrix*, SIAM J. Comput., 8 (1979), pp. 499–507.

[16] N. LITTLESTONE, *Learning when irrelevant attributes abound: A new linear-threshold algorithm*, Machine Learning, 2 (1988), pp. 285–318.

[17] N. LITTLESTONE, *From on-line to batch learning*, in Second Workshop on Computational Learning Theory, Santa Cruz, CA, July 1989, Morgan Kaufmann, Los Altos, CA, pp. 269–284.

[18] W. MAGNUS, A. KARRASS, AND D. SOLITAR, *Combinatorial Group Theory: Presentation of Groups in Terms of Generators and Relations*, John Wiley & Sons, New York, 1966.

[19] T. MITCHELL, *Generalization as search*, Artificial Intelligence, 18 (1982), pp. 203–226.

[20] B. K. NATARAJAN, *Machine Learning: A Theoretical Approach*, Morgan Kaufman, San Mateo, CA, 1991.

[21] L. PITT AND M. K. WARMUTH, *The minimum* DFA *consistency problem cannot be approximated within any polynomial*, Tech. Report UIUCDCS-R-89-1499, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, Feb. 1989.

[22] ———, *The minimum* DFA *consistency problem cannot be approximated within any polynomial*, J. Assoc. Comput. Mach., to appear.

[23] H. SHVAYTSER, *Linear manifolds are learnable from positive examples*, unpublished manuscript, 1988.

[24] V. N. VAPNIK AND A. Y. CHERVONENKIS, *On the uniform convergence of relative frequencies of events to their probabilities*, Theory Probab. Appl., 16 (1971), pp. 264–280.

[25] T. WU J. CHOU AND G. E. COLLINS, *Algorithms for the solution of systems of linear Diophantine equations*, SIAM J. Comput., 11 (1982), pp. 687–708.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 267–280, April 1992
© 1992 Society for Industrial and Applied Mathematics
005

EFFICIENT POINT LOCATION IN A CONVEX SPATIAL CELL-COMPLEX*

FRANCO P. PREPARATA† AND ROBERTO TAMASSIA‡

Abstract. In this paper a new approach is proposed to point-location in a three-dimensional cell-complex $\mathcal{P}$, which may be viewed as a nontrivial generalization of a corresponding two-dimensional technique due to Sarnak and Tarjan. Specifically, in a space-sweep of $\mathcal{P}$, the intersections of the sweep-plane with $\mathcal{P}$ occurring in a given slab, i.e., between two consecutive vertices, are topologically conformal planar subdivisions. If the sweep direction is viewed as time, the descriptions of the various slabs are distinct "versions" of a two-dimensional point-location data structure, dynamically updated each time a vertex is swept. Combining the persistence-addition technique of Driscoll, Sarnak, Sleator, and Tarjan [*J. Comput. System. Sci.*, 38 (1989), pp. 86–124] with the recently discovered dynamic structure for planar point-location in monotone subdivisions, a method with query time $O(\log^2 N)$ and space $O(N \log^2 N)$ for point-location in a convex cell-complex with N facets is obtained.

Key words. point location, convex cell complex, computational geometry, analysis of algorithms

AMS(MOS) subject classifications. 68U05, 68Q25, 68P05, 68P10

1. Introduction. Point-location in three-dimensional space, called spatial point-location, is a natural generalization of the well-known planar point location (see, e.g., [6], [11], [12]). The space is partitioned into polyhedral regions, called *cells*, and the resulting subdivision is frequently referred to as a *cell-complex.* The problem is so stated: Given a cell-complex $\mathcal{P}$ and a query point q, determine the cell of $\mathcal{P}$ containing q.

Unlike its two-dimensional counterpart, spatial point-location has not yet received extensive attention. In all reported research, the cell-complex satisfies some restrictive condition. Cole's Similar Lists method [4] has been applied to the cell complex determined by an arrangement of n planes, and yields query time $O(\log n)$ but uses space $O(n^4/\log n)$. The space bound has been recently improved by Chazelle and Friedman [2] to $O(n^3)$, by a modification of the random sampling technique of Clarkson [3]. Chazelle's earlier Canal Tree technique [1] trades space for query time, and achieves space $O(n^3)$ and query time $O(\log^2 n)$. The same technique can be applied to a convex cell-complex with N facets, yielding query time $O(\log^2 N)$ and space $O(N)$; the cell-complex, however, is subject to the restrictive condition that the vertical dominance relation on the cells be acyclic.

Two recent results can be profitably combined to provide a new attractive approach to spatial point-location: the persistence-addition technique of Driscoll, Sarnak, Sleator, and Tarjan [5] and the dynamic planar point-location technique of

* Received by the editors December 26, 1989; accepted for publication (in revised form) March 26, 1991. An extended abstract of this paper was presented at the 1989 Workshop on Algorithms and Data Structures, Ottawa, Canada, August 1989.

† Department of Computer Science, Brown University, Providence, Rhode Island 02912–1910. The research of this author was supported in part by National Science Foundation grant CCR-89-06419. This research was initiated while the author was with the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

‡ Department of Computer Science, Brown University, Providence, Rhode Island 02912–1910. The research of this author was supported in part by National Science Foundation grant CCR-9007851, by the United States Army Research Office grant DAAL03-91-G-0035, by Office of Naval Research and Defense Advanced Research Projects Agency contract N00014-91-J-4052 and ARPA order 8225, and by Cadre Technologies, Inc.

Preparata and Tamassia [13]. Our result is a new method with query time $O(\log^2 N)$ and space $O(N \log^2 N)$ for a convex spatial cell-complex with N facets.

The general methodology of [5] is designed to add *persistence* to a dynamic linked data structure (referred to as *ephemeral*). In an abstract setting, the ephemeral data structure supports accesses and updates, and each update produces a new version of the structure. Thus the history of the data structure is the sequence of its versions, terminating in the *current* version. Persistence is the ability to access past versions, and it is *full* or *partial* depending upon whether updates are also permitted in past versions or not. In [5] a systematic and efficient technique is presented to transform an ephemeral linked data structure into a persistent one, provided that the ephemeral structure satisfies the weak condition that its nodes have bounded in-degree. This methodology was applied in a companion paper [14] to planar point-location, by viewing one dimension (e.g., the y-direction) as "time," so that the planar subdivision is swept over in time by a horizontal line. The dictionary of the sequence of intersected edges is the ephemeral data structure, a new version of which is created each time a vertex of the subdivision is reached in the sweep. The persistent version of the dictionary becomes therefore the data structure for planar point location; in other words, *static* two-dimensional point-location is modeled by a *partially persistent* one-dimensional dictionary process.

The last observation is the clue to higher-dimensional generalizations. Until recently, however, the obstacle to a three-dimensional generalization was the lack of a suitable (ephemeral) dynamic planar point-location structure. The recent discovery [13] of an efficient such structure for monotone planar subdivisions provides the missing component that, combined with the technique for the addition of persistence, yields the method for point location in a spatial convex cell complex discussed in this paper.

This paper is organized as follows. In §2 we review the separator-tree structure that is the basis of the two-dimensional point-location primitive, as well as the essentials of the persistence-addition technique. Section 3 illustrates the modifications required to adapt the ephemeral data structure to the projected spatial point-location method, described in its most general version in §4. Finally, in §5 we describe the simplifications obtainable when exploiting the specific nature of particular cell-complexes such as Voronoi diagrams and those determined by arrangements of planes.

2. Preliminaries.

2.1. Planar point location. Let $\mathcal{P}$ be a three-dimensional cell complex, i.e., a partition of the three-dimensional space into polyhedra. We say that $\mathcal{P}$ is *convex* if either each of its cells is a convex polyhedron, or at most one is nonconvex (in this case, the nonconvex cell is the complement of a convex polyhedron). Any planar section of $\mathcal{P}$ is a planar convex subdivision (with the analogous exception of at most one nonconvex region), which is a special case of a monotone subdivision [9]. We shall now review the essentials of the dynamic point-location method of Preparata and Tamassia [13] for planar monotone subdivisions, in order to bring into sharper focus the requirements for the addition of persistence.

An *edge* is either a segment or a half-line in the plane (for simplicity, no edge is horizontal). A (polygonal) *chain* is a sequence of edges, so that two consecutive edges share a terminus; it is *monotone* if each horizontal line intersects it in at most a single point. A polygon is *monotone* if its boundary is partitionable into two monotone chains. A subdivision $\mathcal{R}$ is *monotone* if each of its regions is a monotone polygon.

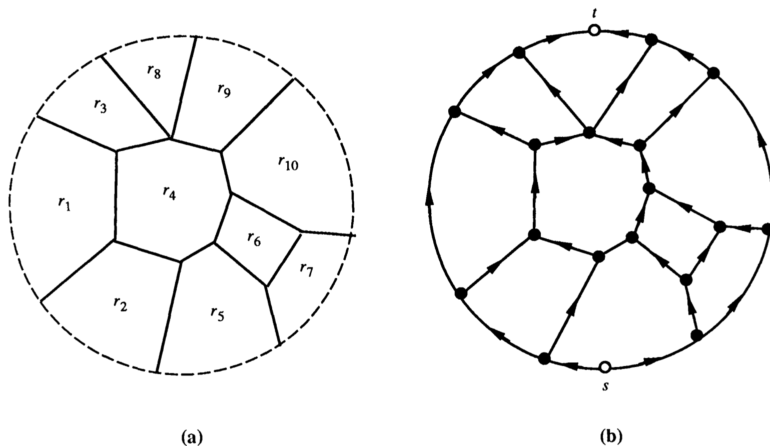


FIG. 1. *Dynamic point location technique:* (a) *monotone subdivision* $\mathcal{R}$ (*the dashed circle represents the line at infinity*); (b) *planar st-graph G associated with* $\mathcal{R}$.

A *separating chain* (or, more succinctly, a *separator*) σ of a monotone subdivision $\mathcal{R}$ is a monotone chain consisting of edges of $\mathcal{R}$, whose extreme points are at infinity. Given two separators σ_1 and σ_2, σ_1 *is to the left* of σ_2 if any horizontal line intersects σ_1 not to the right of σ_2. A *complete family of separators* Σ of $\mathcal{R}$ is a sequence of separators $(\sigma_1, \sigma_2, \cdots, \sigma_t)$ such that σ_i is to the left of σ_{i+1} $(i = 1, \cdots, t-1)$, and every edge of $\mathcal{R}$ belongs to at least one separator $\sigma \in \Sigma$. For a given Σ, each $\sigma \in \Sigma$ is associated with a node (briefly referred to as "node σ") of a balanced binary tree $\mathcal{T}$ (called *separator tree*). An edge e of $\mathcal{R}$ belongs, in general, to a nonempty interval $(\sigma_i, \sigma_{i+1}, \cdots, \sigma_j)$ of separators. However, it is assigned to the least common ancestor σ_k of $(\sigma_i, \cdots, \sigma_j)$ in $\mathcal{T}$, and is called a *proper edge* of σ_k. Correspondingly, *proper* (σ_k) denotes the set of proper edges of σ_k, which are stored in a secondary component (a dictionary) appended to node σ_k of $\mathcal{T}$. Since each edge of $\mathcal{R}$ is stored exactly once, the above data structure has size $O(n)$, where n is the number of vertices of $\mathcal{R}$. It is well known [9] that point location in $\mathcal{R}$ corresponds to traversing a path from the root of $\mathcal{T}$, and performing at each node a point/chain discrimination by means of the secondary component. Thus point-location can be done in $O(\log^2 n)$ time.

The underlying topological structure of a monotone subdivision is a *planar st-graph*, i.e., a planar acyclic digraph with exactly one source (vertex without incoming edges), s, and exactly one sink (vertex without outgoing edges), t, embedded in the plane with vertices s and t on the boundary of the external face. Namely, we associate a monotone subdivision $\mathcal{R}$ with the planar st-graph G defined as follows (see Fig. 1):

1. The vertices of G are the vertices of $\mathcal{R}$, including vertices at infinity determined by edges that are rays, plus vertices s and t corresponding to the points at negative and positive infinity on the vertical axis, respectively.
2. The edges of G are the edges of $\mathcal{R}$, oriented from the lower to the upper endpoint, plus two chains from s to t that connect the vertices at infinity.

There are two main obstacles to the dynamization of $\mathcal{T}$ for an arbitrary family of separators Σ:

1. It may be difficult to rebuild the secondary components of the nodes participating in a rotation of the primary component. Indeed, in the worst case a rotation takes time $\Omega(n)$.
2. The deletion of an edge may break many separators of Σ, with $\Omega(n)$ time used for restructuring the complete family Σ.

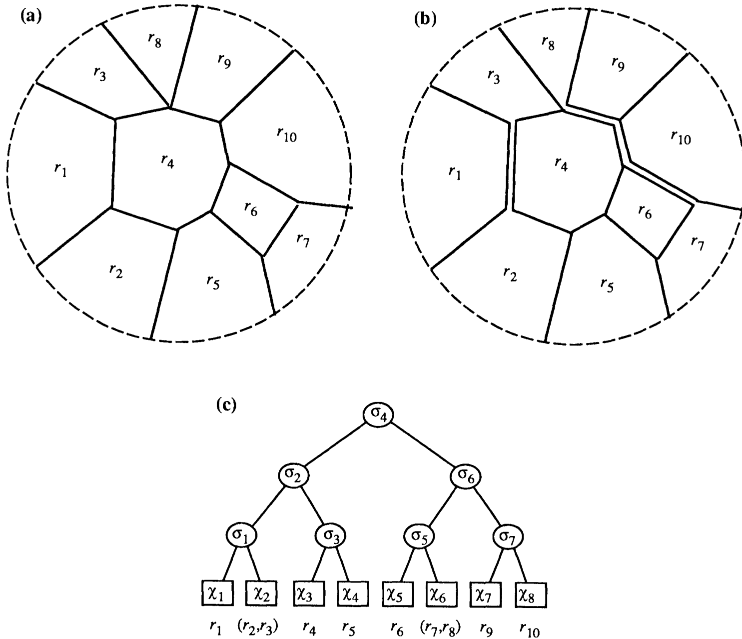


FIG. 2. *Dynamic point location technique:* (a) *monotone subdivision* $\mathcal{R}$; (b) *subdivision* $\mathcal{R}^*$ *and its chains of proper edges;* (c) *separator tree of* $\mathcal{R}^*$.

It has been shown in [13], however, that a modification (consisting of virtually duplicating a subset of edges) transforms $\mathcal{R}$ into a new subdivision $\mathcal{R}^*$ that admits a unique complete family of separators Σ^* such that, for any $\sigma \in \Sigma^*$, *proper* (σ) consists of a single chain. This permits an $O(\log n)$-time execution of a rotation in the corresponding separator tree $\mathcal{T}^*$ by standard operations on concatenable queues. Also, the restructuring of Σ^* after a deletion can be done with $O(\log n)$ split/splice operations on the chains of proper edges. The family Σ^* enjoying these properties is associated with the following total order "$\prec$" on the regions of $\mathcal{R}$ (see Fig. 2):

1. r_1 *is below* r_2 $(r_1 \uparrow r_2)$ if there is a monotone chain in $\mathcal{R}$ from the highest vertex of r_1 to the lowest vertex of r_2 (such chain corresponds to a directed path in digraph G);
2. r_1 *is to the left of* r_2 $(r_1 \rightarrow r_2)$ if there is sequence of regions $r_1 = r'_1, r'_2, \cdots, r'_s = r_2$ such that r'_i and r'_{i+1} share an edge and r'_i is to the left of it $(i = 1, \cdots, s-1)$. Partial orders "$\uparrow$" and "$\rightarrow$" are shown to be complementary, so that their union is the desired total order "$\prec$";
3. The members of any maximal sequence of regions $r_1 \prec r_2 \prec \cdots \prec r_s$, consecutive in "$\prec$" and such that $r_i \uparrow r_{i+1}$ $(i = 1, \cdots, s-1)$ are merged into a generalized region called *cluster* by duplicating the vertices and, when they exist, the edges of the (unique) monotone chain connecting the highest vertex of r_i to the lowest vertex of r_{i+1}, for $i = 1, \cdots, s-1$. This gives the subdivision $\mathcal{R}^*$. Σ^* is the unique complete family of separators for $\mathcal{R}^*$, each member of which separates contiguous clusters.

By using the correspondence between monotone subdivisions and planar st-graphs, the topological underpinning of the total order $\prec$ on the regions of $\mathcal{R}$ can be found in the theory of planar st-graphs and planar lattices [8], [10], [15].

We shall use, where appropriate, a string notation to illustrate the order $\prec$, so

that AB denotes that the partial subdivision described by the string A precedes in $\prec$ the one described by B.

The repertory of updates considered in [13] consists of: insertion/deletion of a vertex internal to a segment, and insertion/deletion of a chain of edges between two vertices, under the condition that monotonicity be preserved. For these specifications the following performance is achieved.

THEOREM 2.1 [13]. *Let $\mathcal{R}$ be a monotone subdivision with n vertices. There exists a dynamic point-location data structure with space $O(n)$ and query time $O(\log^2 n)$, which allows for insertion/deletion of a vertex in time $O(\log n)$ and insertion/deletion of a k-edge chain in time $O(\log^2 n + k)$.*

2.2. The technique for the addition of persistence. We now review the essentials of the technique of Driscoll, Sarnak, Sleator, and Tarjan [5] to add persistence to a dynamic linked data structure, which supports access and update operations. (A conventional dynamic data structure is called *ephemeral* if its instantiation preceding an update is not recoverable after the execution of the update.) The m updates are chronologically numbered from 1 to m, and the ith update generates version i of the ephemeral data structure. A *fully persistent* structure supports both accesses and updates to any of its versions; a *partially persistent* structure supports accesses to any of its versions, but updates only to its most recent (*current*) version. In this paper we are concerned exclusively with partial persistence.

The persistent structure embeds all versions of the ephemeral structure, so that access to any of them can be effectively simulated. Specifically, an access is the traversal of a path in an ephemeral version; the simulation of this access is possible if the persistent structure contains an image of this path. This is effectively accomplished by replacing each outgoing pointer of the ephemeral structure with a bundle of instantiations of that pointer in the persistent structure, each one time-stamped with the index of the update that established it, and ordered accordingly. Thus simulation of an access to version j is effectively accomplished by following at each step the appropriate pointer with maximum time-stamp not exceeding j. Arbitrarily large bundle size, however, negatively affects access time and storage. To avoid this shortcoming, Driscoll, Sarnak, Sleator, and Tarjan enforce a bound K on the number of pointers issuing from any given node and introduce the device of *limited node copying*, to be briefly outlined below.

A new copy v' of a node v is to be created at update j either when an information field of v is modified or a pointer field of v is modified, and, in either case, the corresponding update would cause the number of pointers issuing from v to exceed the bound K. In both cases, node v is declared *dead*, while the *live* node v' is time-stamped j; in addition, node v' must be correctly linked within the structure. This is facilitated if each pointer from live node to live node is paired with a reverse pointer (which need not be time-stamped). Specifically, we have the following actions, which implement copying of node v following an update of field f at time j:

1. All fields of v, but f (which receives the updated value), are copied into v'. Among these, for each pointer field to a live node w, the corresponding reverse pointer in w is switched from v to v'.
2. Each reverse pointer from v to some (live) w is suppressed, and (if the time-stamp of the direct pointer from w to v is less than j) a new pointer time-stamped j is established from w to v' (along with the corresponding reverse pointer).

The actions of step 2 deserve further discussion. First, to ensure that copying of

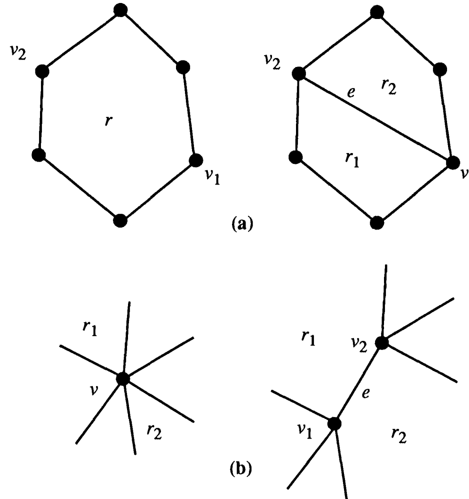


FIG. 3. (a) *Operations Insert and Delete.* (b) *Operations Expand and Contract.*

a node be accomplished in $O(1)$ time, it is sufficient to require that the in-degree of a node in the ephemeral structure be bounded by a constant. (Under this condition, only a bounded number of reverse pointers must ever be suppressed.) Second, step 2 introduces additional pointers outgoing from an existing node w and may cause the total number of outgoing pointers of w to exceed bound K. Consequently, node copying may "bubble up" the linked data structure. However, amortized over a sequence of updates, Driscoll, Sarnak, Sleator, and Tarjan [5] show that there are only $O(1)$ nodes copied per update.

This is summarized as follows.

THEOREM 2.2 [5]. *If an ephemeral data structure has nodes of constant bounded in-degree, then the structure can be made partially persistent at an amortized space cost of* $O(1)$ *per update step and a constant factor in the amortized time per operation.*

Considerable simplifications arise when the ephemeral data structure is a red-black tree [7]. In this case the node in-degree is at most 1, and rebalancing after an insertion/deletion requires $O(1)$ rotations.

3. The ephemeral planar point location structure. In this section we describe algorithms and data structures designed to support the following repertory of update operations on a monotone subdivision $\mathcal{R}$ (see Fig. 3).

Insert$(e, r, v_1, v_2; r_1, r_2)$: Insert edge e between vertices v_1 and v_2 inside region r, which is decomposed into regions r_1 and r_2 to the left and right of e, respectively.

Delete$(e, v_1, v_2, r_1, r_2; r)$: Remove edge e between vertices v_1 and v_2 and merge into region r the two regions r_1 and r_2 formerly to the left and right of e, respectively.

Expand$(e, v, r_1, r_2; v_1, v_2)$: Expand vertex v into vertices v_1 and v_2 connected by edge e, which has regions r_1 and r_2 to its left and right, respectively.

Contract$(e, r_1, r_2, v_1, v_2; v)$: Contract edge e between vertices v_1 and v_2 into vertex v. Regions r_1 and r_2 are those formerly to the left and right of e, respectively.

The present techniques represent modifications of those described in [13], where only operations *Insert* and *Delete* are supported, and nodes of the data structures do

not have the now required bounded in-degree. Note that, in terms of the underlying topological structure of a monotone subdivision discussed in §2.1, operations (*Insert*, *Expand*) and (*Delete*, *Contract*) form dual pairs, the first term acting on the primal graph (coincident with the subdivision), the second term acting on its dual graph. The data structure, however, will not explicitly reflect this duality; indeed, the vertices of $\mathcal{R}$ play a central role, since the geometry of $\mathcal{R}$ is determined by the coordinates of its vertices.

In the original data structure [13], a vertex was pointed to by all of its incident edges and by all regions having it as the lowest/highest point. Since a vertex can receive $\Omega(n)$ such pointers, the original representation both prevents an efficient implementation of operations *Expand*/*Contract* and violates the bounded in-degree requirement. Therefore, we propose the following modified structure, which comprises a main constituent, called the "augmented separator tree," and an auxiliary constituent, called the "dictionary."

The *dictionary* has a record for each vertex, edge, and region of $\mathcal{R}$, as follows:

1. The record of vertex v stores the coordinates of v and pointers to two balanced binary search trees, denoted $in(v)$ and $out(v)$, and called the *incidence trees* of vertex v. The leaves of tree $in(v)$ represent the incoming edges of v (i.e., the edges (u, v) with $y(u) < y(v)$), and the internal nodes of $in(v)$ represent the regions of $\mathcal{R}$ whose highest vertex is v, where the in-order sequence of the nodes of $in(v)$ corresponds to the counterclockwise order of the corresponding edges and regions around v. Each node of $in(v)$ has a pointer to the record of the corresponding edge or region. Tree $out(v)$ is similarly defined with respect to the outgoing edges of v. Also, the record of vertex v contains a pointer to the representative node of v in the (secondary component of the) augmented separator tree, to be described below.
2. The record of edge $e = (u, v)$ stores pointers to the representative nodes of e in the trees $in(v)$ and $out(u)$. Also, the record of e stores pointers to the two representative nodes of e in the (secondary component of the) augmented separator tree, to be described below.
3. The record of region r stores pointers to the representative nodes of r in the trees $in(v)$ and $out(u)$, where u and v are the lowest and highest vertex of r, respectively. Also, the record of region r stores a pointer to the representative node of r in the (secondary component of the) augmented separator tree, to be described below.

The dictionary allows us to find the endpoints of an edge and the lowest and highest vertices of a region in $O(\log n)$ time. The dynamic maintenance of the dictionary can be performed in $O(\log n)$ time per update, and will not be explicitly discussed. Note that an *Expand* or a *Contract* operation corresponds to performing $O(1)$ split/splice operations in the incidence trees.

The *augmented separator tree*, denoted $\mathcal{T}^*$, has a primary and secondary component. The *primary component* is a balanced separator tree for $\mathcal{R}^*$, i.e., each of its leaves is associated with a region of $\mathcal{R}^*$ (a cluster of $\mathcal{R}$), and each of its internal nodes is associated with a separator of $\mathcal{R}^*$. The left-to-right order of the leaves of the primary component of $\mathcal{T}^*$ corresponds to the order $\prec$ on the regions of $\mathcal{R}^*$. The *secondary component* is a collection of balanced search trees, each associated with a node of $\mathcal{T}^*$:

1. Each internal node σ of $\mathcal{T}^*$ points to a balanced search tree $proper(\sigma)$ representing the chain $proper(\sigma)$, and to a balanced search tree $double(\sigma)$ asso-

ciated with the proper edges of σ that do not form a channel (called *double edges*). Whereas in [13] the detailed organization of the concatenable queues describing a separator was left unspecified, here the bounded in-degree requirement can be achieved as follows, without affecting the efficiency of point-location queries. Tree $proper(\sigma)$ is organized so that each leaf represents an edge, and each internal node represents a vertex, where the in-order sequence of the nodes corresponds to traversing the chain of proper edges of σ from bottom to top. Each node of $proper(\sigma)$ points to the record of the corresponding vertex or edge in the dictionary. Tree $double(\sigma)$ is organized so that each node represents an edge, and the in-order sequence of nodes corresponds to the bottom-to-top subsequence of the double edges of $proper(\sigma)$. Note that each edge e of $\mathcal{R}$ has exactly two representative nodes in the secondary components of the augmented separator tree $\mathcal{T}^*$.

2. Each leaf χ of $\mathcal{T}^*$ (i.e., the leaf representing cluster χ) points to a balanced search tree $regions(\chi)$ associated with the sequence of regions that form cluster χ, in bottom-to-top order.

The following theorem summarizes the properties of the ephemeral structure.

THEOREM 3.1. *Let $\mathcal{R}$ be a monotone planar subdivision with n vertices. There exists a dynamic point-location data structure for $\mathcal{R}$ that* (i) *has bounded record in-degree,* (ii) *uses $O(n)$ space, and* (iii) *supports queries and update operations Insert, Expand, Delete, and Contract, each in $O(\log^2 n)$ time.*

Proof. (i) It is straightforward to verify that the above data structure has records with bounded in-degree. (ii) The space used is $O(1)$ per vertex, edge, and region. Hence, by Euler's formula, the total space requirement is $O(n)$. (iii) The algorithms for queries and operations *Insert* and *Delete* are essentially as described in [13], with the following minor modifications to take into account the variations of the data structure. In the query algorithm, tree $proper(\sigma)$ is used to discriminate the query point with respect to separator σ. Once the cluster χ containing the query point has been determined, finding the region of χ containing the query point takes time $O(\log^2 n)$, since the lowest and highest vertices of each region in $regions(\chi)$ are retrieved in $O(\log n)$ time through the dictionary. As shown in [13], the *Insert* and *Delete* algorithms determine a partition of the subdivision $\mathcal{R}$ into $O(1)$ partial subdivisions, called *canonical components*, each represented by an augmented separator-tree. The augmented separator-trees of the canonical components are obtained by splitting the augmented separator tree of $\mathcal{R}$. The canonical components are subsequently reassembled, to yield the updated subdivision, and their augmented separator-trees are appropriately spliced to yield the updated augmented separator-tree of $\mathcal{R}$. Each split/splice operation of the augmented separator-tree is achieved with $O(\log n)$ rotations of the primary separator-tree, for a total time complexity $O(\log^2 n)$.

Now, we discuss operation $Expand(e, v, r_1, r_2; v_1, v_2)$. (Operation *Contract* is symmetric.) We consider two cases, depending on the relative order of r_1 and r_2 in $\prec$ prior to the execution of *Expand*:

Case 1. $r_2 \prec r_1$ (see Fig. 4(a)). Note that in this case we must also have $r_2 \uparrow r_1$. The sequence of regions before the expansion, sorted according to the order $\prec$, can be written as a string of the form:

$$A\alpha - - r_2 B\beta — \gamma C r_1 - - \delta D.$$

Here, Greek letters denote clusters or portions of clusters, capital letters denote subsequences of regions that contain complete clusters, "—" denotes a channel, and "- -" denotes a potential channel. (Some of the letters may denote empty subsequences.)

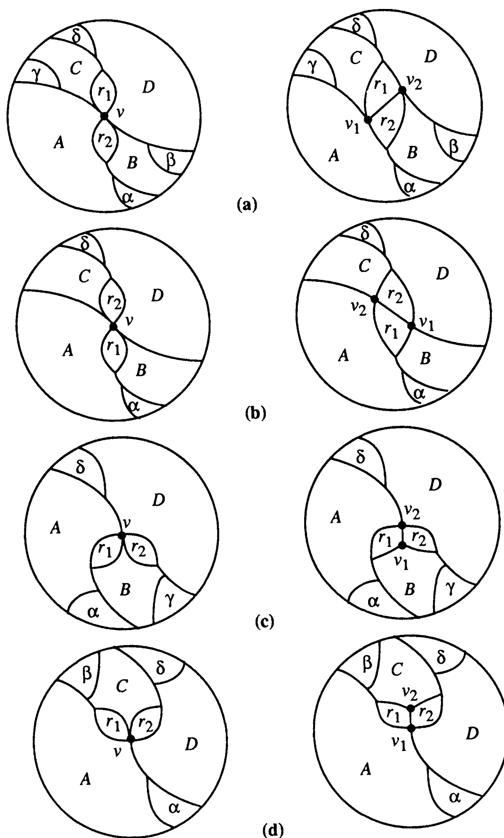


FIG. 4. *Four cases for operation Expand*$(e, v, r_1, r_2; v_1, v_2)$*:* (a) $r_2 \uparrow r_1$*;* (b) $r_1 \uparrow r_2$*;* (c) $r_1 \rightarrow r_2$ *and same lowest vertex for* r_1 *and* r_2*;* (d) $r_1 \rightarrow r_2$ *and same highest vertex for* r_1 *and* r_2.

After the expansion, the new sequence of regions is:

$$A\alpha - - \gamma C r_1 r_2 B \beta - - \delta D.$$

Case 2. $r_1 \prec r_2$ (see Figs. 4(b)–(d)). The sequence of regions before the expansion is now of the form:

$$A\alpha - - r_1 - - \beta B — C\gamma - - r_2 - - \delta D,$$

where some of the letters may denote empty subsequences. The expansion leaves the sequence of regions unaltered.

Hence, operation *Expand*$(e, v, r_1, r_2; v_1, v_2)$ can also be performed by decomposing the subdivision $\mathcal{R}$ into its canonical components A, B, C, D, α, β, γ, δ, r_1, and r_2, which are subsequently reassembled in the new order to yield the updated $\mathcal{R}$. We conclude that operation *Expand* and its symmetric *Contract* take $O(\log^2 n)$ time. □

It is important to observe that the dynamic point-location data structure of Theorem 3.1 depends on the topology, and not on the specific geometry, of the subdivision. This is formally expressed by the following straightforward lemma.

LEMMA 3.2. *Let* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *be monotone subdivisions whose associated planar st-graphs* G_1 *and* G_2 *are isomorphic. A dynamic point-location data structure for* $\mathcal{R}_1$ *(as discussed in Theorem* 3.1*) can be used for dynamic point-location in* $\mathcal{R}_2$ *after implementing the appropriate modifications of values of the vertex coordinates.*

Note, however, that for a given monotone subdivision $\mathcal{R}$, there are several versions of the dynamic point-location data structure, corresponding to equivalent versions of the primary separator-tree and of the secondary trees.

4. Spatial point-location. Let $\mathcal{P}$ be a convex cell-complex with n vertices and N facets. Note that both n and the number of edges of $\mathcal{P}$ are $O(N)$. The z-coordinates of the vertices are denoted $z_1, \cdots, z_n$, from bottom to top. Let $\mathcal{P}(z)$ be the intersection of $\mathcal{P}$ with the plane $\pi(z)$ parallel to the x and y axes and at height z. It is easy to verify that $\mathcal{P}(z)$ is a convex subdivision. Also, we define $G(z)$ as the planar st-graph associated with $\mathcal{P}(z)$.

By viewing z as a measure of "time" we consider the process of making plane $\pi(z)$ sweep the cell complex $\mathcal{P}$. While the geometry of $\mathcal{P}(z)$ continuously evolves in time, its topology changes only when plane $\pi(z)$ goes through a vertex v of $\mathcal{P}$. This is formalized as follows.

LEMMA 4.1. *For z', z'' such that $z_i < z', z'' < z_{i+1}$ the digraphs $G(z')$ and $G(z'')$ are isomorphic.*

Proof. The vertices and edges of $\mathcal{P}(z)$ are the intersections of the edges and facets of $\mathcal{P}$ with $\pi(z)$, respectively. Let $f(z)$ be the edge of $\mathcal{P}(z)$ generated by the intersection of facet f with $\pi(z)$. The slope of $f(z)$ in the plane $\pi(z)$ is the same for all z such that $\pi(z)$ intersects f. Since there are no vertices of $\mathcal{P}$ in the region of space $z' < z < z''$, we conclude that the digraphs $G(z')$ and $G(z'')$ are isomorphic. □

By Lemmas 3.2 and 4.1 the same point-location data structure can be used for all query points whose z-coordinate is in the open range (z_i, z_{i+1}), provided the x and y coordinates of the vertices are expressed as (linear) functions of z.

An *event* occurs when plane $\pi(z)$ goes through a vertex v of $\mathcal{P}$, and results in updating the subdivision $\mathcal{P}(z)$ (see Fig. 5). Let z_- and z_+ be "instants" immediately preceding and following z. The transformation from $\mathcal{P}(z_-)$ to $\mathcal{P}(z)$ consists of contracting a subgraph G_- into vertex v. The vertices (respectively, the edges) of G_- are those associated with the edges (respectively, the facets) of $\mathcal{P}$ whose highest vertex is v. Conversely, the transformation from $\mathcal{P}(z)$ to $\mathcal{P}(z_+)$ consists of expanding vertex v into a subgraph G_+. The vertices (respectively, the edges) of G_+ are those associated with the edges (respectively, the facets) of $\mathcal{P}$ whose lowest vertex is v.

The following lemma shows that the above contractions/expansions of subgraphs into/from vertices can be performed by a sequence of elementary contractions/expansions of edges into/from vertices.

A subgraph S of a planar st-graph G is said to be *contractible* if S is vertex-induced and the graph G_S obtained from G by contracting S into a single vertex is itself a planar st-graph.

LEMMA 4.2. *Let G be a planar st-graph, and S a contractible subgraph of G with m edges. There exists a sequence of m elementary updates, each a deletion or a contraction, that transforms G into G_S.*

Proof. Consider in turn each edge e of S, and recall that any edge of a planar st-graph is either removable or contractible [15]. If e is contractible, then contract it, otherwise remove it. We claim that this process will correctly produce graph G_S. Indeed, the process would fail if the current edge $e = (v', v'')$ is a bridge (separation edge) of the graph to which S has currently transformed, but is not contractible (see Fig. 6). This implies that the graph to which G has currently transformed has a directed path from v' to v'' that contains at least one vertex w not in S. Hence, G_S has a directed cycle containing vertex w. This contradiction establishes the

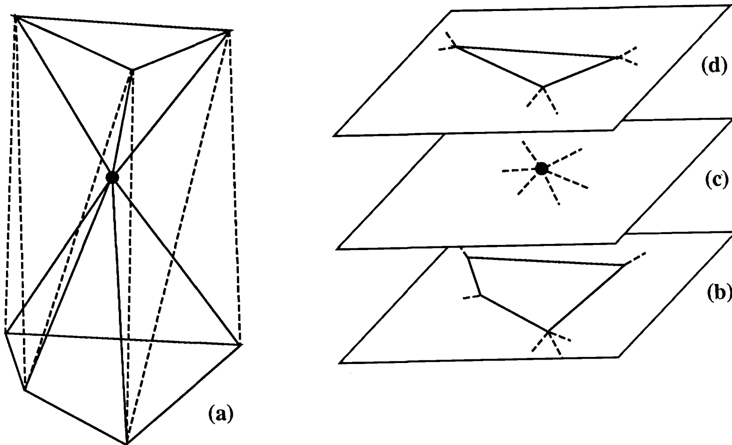


FIG. 5. (a) *Cell complex.* (b) *Graph* $G(z_-)$. (c) *Graph* $G(z)$. (d) *Graph* $G(z_+)$.

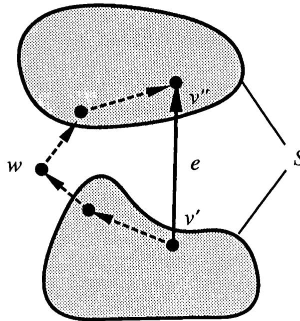


FIG. 6. *Example of contraction of a subgraph S of a planar st-graph by means of elementary deletions and contractions of edges. A contradiction is achieved if edge e is a bridge of the current S but is not contractible.*

claim. □

The proof of the above lemma shows that the contraction is executable in total time $m\times$(update time). The symmetric pair (*Insertion, Deletion*) and their dual pair (*Expansion, Contraction*) readily establish the following complementary result.

LEMMA 4.3. *Let G be a planar st-graph, and S a contractible subgraph of G with m edges. There exists a sequence of m elementary updates, each an insertion or an expansion, that transforms* G_S *into G.*

THEOREM 4.4. *Space-sweep of a convex cell complex* $\mathcal{P}$ *with N facets can be performed in* $O(N\log^2 N)$ *time so that at any time point-location queries can be answered in* $O(\log^2 N)$ *time and the space requirement is* $O(N)$.

Proof. By Lemmas 3.2 and 4.1 the space-sweep process goes through the $2n+1$ topologically different subdivisions $\{\mathcal{P}(z_j) : j = 1,\cdots,n\}$ and $\{\mathcal{P}(z'_j) : j = 0,\cdots,n;\ z_j < z'_j < z_{j+1};\ z_0 = -\infty;\ z_{n+1} = +\infty\}$. By Lemmas 4.2–4.3 the updates of the ephemeral point-location data structure are no more than $2N$. By Theorem 3.1 each such update can be processed in time $O(\log^2 N)$. □

The (ephemeral) dynamic point-location data structure of Theorem 3.1 verifies the conditions for the applicability of the technique of [5], and can therefore be converted

into a partially persistent one that supports queries in past versions. Therefore, we obtain the central result of this paper in the following theorem.

THEOREM 4.5. *Let $\mathcal{P}$ be a convex spatial cell-complex with N facets. Point-location in $\mathcal{P}$ can be performed in time $O(\log^2 N)$ using an $O(N \log^2 N)$-space data structure that can be constructed in time $O(N \log^2 N)$.*

The result of Theorem 4.5 can be easily extended to a nonconvex spatial cell-complex $\mathcal{P}$ such that each subdivision $\mathcal{P}(z)$ $(-\infty < z < +\infty)$ is connected and monotone.

5. Adding persistence to special cell-complexes. In this section we illustrate the simplifications of the general technique which occur when exploiting special properties of the cell-complex and the prior knowledge of the problem instance before executing the space-sweep. (Note that the latter feature is, in general, a potential source of simplification in persistence-addition techniques.) Specifically, we shall consider Voronoi diagrams and the cell-complexes determined by arrangements of planes. We shall show that the update of the order on the set of regions of subdivision $\mathcal{P}(z)$ (which defines the ephemeral point-location data structure) requires only insertions and deletions, but no substantial restructurings (corresponding to the swaps of substrings occurring in the general case). We begin with Voronoi diagrams.

THEOREM 5.1. *Let $\mathcal{P}$ be the cell-complex induced by the Voronoi diagram of n sites in three-dimensional space. Processing an event in the space-sweep of $\mathcal{P}$ takes $O(\log n)$ time.*

Proof. Excluding degeneracies, every vertex of $\mathcal{P}$ has degree four. Hence processing an event consists of expanding a vertex v into a triangle r, or of contracting r into v. Instead of simulating such transformation by means of elementary updates, we consider directly its effect of the ordering of the regions of $\mathcal{P}(z)$. We use the extension of the order $\prec$ to the set of vertices, edges, and regions of a monotone subdivision introduced in [15], and the local characterization of such ordering given in [16]. First, consider the expansion of vertex v into a triangle r. The update of the extended order is performed by simply replacing v with r and its vertices and edges. Hence, the sequence of the regions is modified by the insertion of r. Symmetrically, the contraction of r into v causes the deletion of r from the sequence of regions.

Since the primary separator-tree is modified only by insertions and deletions, we implement it as a red-black tree [7], so that rebalancing after an insertion or deletion is done with $O(1)$ rotations. The time bound follows from the fact that a rotation of the primary separator-tree takes $O(\log n)$ time, due to the split/splice operations on the trees of proper edges attached to the nodes involved in the rotation. □

The technique for arrangements of planes uses a simpler ephemeral planar point-location structure, especially designed for space-sweep of arrangements.

THEOREM 5.2. *Let $\mathcal{P}$ be the cell-complex induced by an arrangement of n planes in three-dimensional space. Processing an event in the space-sweep of $\mathcal{P}$ takes $O(\log n)$ time and involves $O(1)$ changes of pointers.*

Proof. For an arbitrary z, let $l_1, l_2, \cdots, l_n$ be the intersections of the planes of the arrangement with the sweep-plane $\pi(z)$. On this plane we establish Cartesian axes x and y, which are projections according to z of the homologous axes of the space. Let x_0 be an abscissa such that, for $j = 1, \cdots, n, x_0 \leq x_j$, where x_j is the abscissa of the intersection of l_j with the x-axis. We denote by π_j the half-plane determined by l_j and not containing point $(x_0, 0)$. With each region r of the planar arrangement $\mathcal{P}(z)$ we associate an integer *weight* $w(r) \in \{0, 1, \cdots, n\}$, denoting the number of half-planes in $\{\pi_1, \cdots, \pi_n\}$ containing r. A complete family of n edge-disjoint separators for $\mathcal{P}(z)$ is

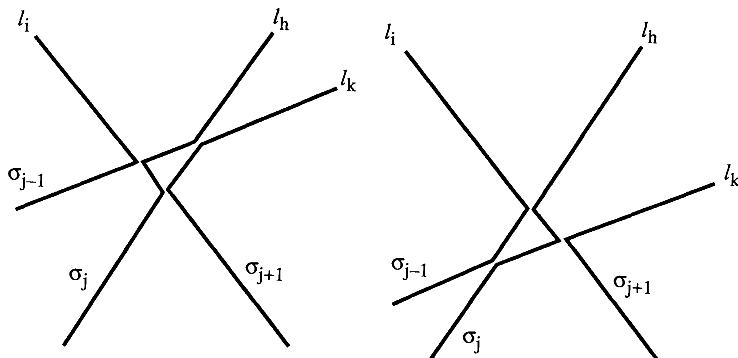


FIG. 7. *Update of the secondary components for the cell-complex induced by an arrangement of planes.*

obtained by defining separator σ_i $(i = 1, \cdots, n)$ as the monotone chain leaving to its left the regions of weight less than or equal to $(i-1)$ and to its right the regions of weight greater than or equal to i. (Note that this family of separators is, in general, not associated with the order $\prec$ defined in §2.1.)

When processing an event, the primary component of the separator tree undergoes no modification. Updates occur only in the secondary components. Specifically, an event point z of the space-sweep corresponds to a point in space where three planes meet. Let l_i, l_h, and l_k be their intersections with $\mathcal{P}(z)$ and let σ_{j-1}, σ_j, and σ_{j+1} be the three separators sharing their common intersection. Suppose that, prior to the update, σ_{j-1} does not contain any portion of l_h; then, the update consists of deleting a segment of l_h from σ_{j+1}, inserting one such segment into σ_{j-1}, and exchanging the order of segments of l_i and l_k in σ_j (see Fig. 7). (A symmetric action takes place when σ_{j+1} contains no portion of l_h.) Assuming that the triplet (l_i, l_h, l_k) corresponding to z has been precomputed, the update of the secondary components, globally involving a bounded number of pointer changes, is executed in time $O(\log n)$. □

Theorems 5.1 and 5.2 lead to the following corollary.

COROLLARY 5.3. *Let $\mathcal{P}$ be the cell-complex in three-dimensional space induced either by the Voronoi diagram of n sites or by an arrangement of n planes. Point-location in $\mathcal{P}$ can be performed in time $O(\log^2 N)$ using an $O(N)$-space data structure that can be constructed in time $O(N \log N)$, where N is the number of facets of $\mathcal{P}$ (N is $O(n^2)$ for the Voronoi diagram and $O(n^3)$ for the arrangement of planes.)*

Acknowledgment. We would like to thank Steven Fortune for useful discussions.

REFERENCES

[1] B. CHAZELLE, *How to search in history*, Inform. and Control, 64 (1985), pp. 77–99.
[2] B. CHAZELLE AND J. FRIEDMAN, *A deterministic view of random sampling and its use in geometry*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, 1988, pp. 539–548.
[3] K. L. CLARKSON, *New applications of random sampling in computational geometry*, Discrete & Comput. Geom., 2 (1987), pp. 195–222.
[4] R. COLE, *Searching and storing similar lists*, J. Algorithms, 7 (1986), pp. 202–220.
[5] J. R. DRISCOLL, N. SARNAK, D. D. SLEATOR, AND R. E. TARJAN, *Making data structures persistent*, J. Comput. System Sci., 38 (1989), pp. 86–124.

[6] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, New York, 1987.

[7] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, in Proc. 19th IEEE Symposium on Foundations of Computer Science, 1978, pp. 8–21.

[8] D. KELLY AND I. RIVAL, *Planar lattices*, Canad. J. Math. 27 (1975), pp. 636–665.

[9] D. T. LEE AND F. P. PREPARATA, *Location of a point in a planar subdivision and its applications*, SIAM J. Comput., 6 (1977), pp. 594–606.

[10] A. LEMPEL, S. EVEN, AND I. CEDERBAUM, *An algorithm for planarity testing of graphs*, Internat. Symposium on Theory of Graphs, Gordon and Breach, New York, 1967, pp. 215–232.

[11] F. P. PREPARATA, *Planar point location revisited: a guided tour of a decade of research*, Internat. J. Found. Comput. Sci. 1 (1990), pp. 71–86.

[12] F. P. PREPARATA AND M. I. SHAMOS, *Computational geometry*, Springer-Verlag, New York, 1985.

[13] F. P. PREPARATA AND R. TAMASSIA, *Fully dynamic point location in a monotone subdivision*, SIAM J. Comput. 18 (1989), pp. 811–830.

[14] N. SARNAK AND R. E. TARJAN, *Planar point location using persistent search trees*, Comm. ACM, 29 (1986), pp. 669–679.

[15] R. TAMASSIA AND F. P. PREPARATA, *Dynamic maintenance of planar digraphs, with applications*, Algorithmica, 5 (1990), pp. 509–527.

[16] R. TAMASSIA AND J. S. VITTER, *Parallel transitive closure and point location in planar structures*, SIAM J. Comput., 20 (1991), pp. 708–725.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 281–294, April 1992

© 1992 Society for Industrial and Applied Mathematics
006

A HEURISTIC OF SCHEDULING PARALLEL TASKS AND ITS ANALYSIS*

QINGZHOU WANG† AND KAM HOI CHENG†

Abstract. This paper investigates the problem of nonreconfigurable, nonpreemptive scheduling of parallel tasks in a homogeneous system of processors. Given a task precedence graph where each task T may be executed on up to $\delta(T)$ processors, the problem is to find a schedule such that the makespan is minimized. A scheduling policy is nonreconfigurable if when more processors become available after a task T has been scheduled on p processors, $p < \delta(T)$, they cannot join the execution of task T. It is nonpreemptive if each task is executed without interruption until completion once the task has started execution. The problem of scheduling parallel tasks is NP-hard. The performance ratio of the makespan produced by the original list scheduling algorithm proposed by Graham [*SIAM J. Appl. Math.*, 17 (1969), pp. 416–429] to the optimal makespan is $\Delta + \frac{m-\Delta}{m}$ [*Inform. Process. Lett.*, 37 (1991), pp. 291–297], where m and Δ are the number of processors in the system and the maximum degree of parallelism in any task, respectively. To provide a performance improvement, the *Earliest Completion Time* (ECT) algorithm is proposed; it is also shown that its performance ratio is bounded by $3 - \frac{2}{m}$. In addition, an example is presented to demonstrate that the bound is at least 2.5.

Key words. heuristic, NP-hard, parallel task, scheduling, task precedence graph

AMS(MOS) subject classifications. 68Q25, 90B10

1. Introduction. Let $\mathcal{P} = \{P_i : i = 1, \cdots, m\}$ be m identical processors, and let $\mathcal{T} = \{T_j : j = 1, \cdots, n\}$ be n tasks. For two tasks $T_j, T_{j'} \in \mathcal{T}$, $T_j \prec T_{j'}$ means that the task $T_{j'}$ cannot be started until the task T_j has been completed. For each task $T_j \in \mathcal{T}$, it may be scheduled to run on p_j processors where $1 \le p_j \le \delta_j$, and δ_j is T_j's maximum degree of parallelism. If a task T_j is scheduled to run on p_j processors, then p_j is called the *scheduled parallelism* of T_j. For each task T_j, τ_j defines the size of the task and if its scheduled parallelism is p_j, the execution time required will be τ_j/p_j.

In this paper, we are concerned with *nonreconfigurable* and *nonpreemptive* scheduling policies. This is based on the postulation that altering the degree of parallelism of a task during its execution or task preemption requires substantial overhead. For a task T_j in any schedule S, we denote its scheduled *start* time $b_S(T_j)$, *completion* time $c_S(T_j)$, and scheduled parallelism $p_S(T_j)$. A schedule is *feasible* if the scheduled parallelism of each task is no greater than its maximum degree of parallelism, it requires no more than m processors at any moment, and it preserves the precedence relationships among tasks, i.e.,

$$(1.1)\qquad \begin{aligned} &1 \le p_S(T_j) \le \delta_j \quad \forall T_j \in \mathcal{T};\\ &\sum_{\substack{1\le j\le n\\ b_S(T_j)\le t< c_S(T_j)}} p_S(T_j) \le m \quad \forall t \in [0,\infty);\\ &T_{j'} \prec T_{j''} \Rightarrow c_S(T_{j'}) \le b_S(T_{j''}) \quad \forall T_{j'}, T_{j''} \in \mathcal{T}. \end{aligned}$$

In a feasible schedule S, the ready time of a task T_j, $r_S(T_j)$, is defined to be the latest completion time of all T_j's predecessors, i.e., $r_S(T_j) = \max_{T_{j'}\prec T_j} c_S(T_{j'})$. If a task T_j does not have any predecessor in $\mathcal{T}$, then $r_S(T_j) = 0$. The performance of a feasible schedule S is measured by its makespan, $M_S(\mathcal{T}, \mathcal{P}) = \max_{T_j\in\mathcal{T}} c_S(T_j)$. The objective

*Received by the editors July 23, 1990; accepted for publication (in revised form) May 10, 1991. This research is based in part upon work supported by the Texas Advanced Research Program under grant 1028-ARP.

†Computer Science Department, University of Houston, Houston, Texas 77204-3475.

of the scheduling problem is to find a feasible schedule such that the makespan is minimized. Let $M_{OPT}(\mathcal{T},\mathcal{P})$ be the makespan of an optimal schedule.

The problem of task scheduling has been studied extensively and has many variations [8]. For sequential tasks, i.e., tasks which can be executed on only one processor at a time, Graham [7] has studied the scheduling problem of identical processors, while Cho and Sahni [4] and Gonzales, Ibarra, and Sahni [6] have studied the scheduling problem of uniform (not identical) processors. The introduction of parallel tasks into the scheduling problem can be found in [1]–[3], [5], and [9], and a recent survey on the multiprocessor scheduling problem may be found in [10]. Blazwicz, Drabowski, and Weglarz [2] have proposed polynomial time optimal algorithms for two special cases of the independent parallel task scheduling problem. Chen and Lai [3] have studied the scheduling problem of independent parallel tasks on a hypercube system that require the size of each task to be a power of two, and use the linear speedup assumption. At about the same time, Krishnamurti and Ma [9] were investigating the problem of independent parallel tasks using the less than linear speedup assumption. Du and Leung [5] have proved that except in some special cases, both the preemptive and nonpreemptive parallel task scheduling problems are strongly NP-hard. Recently, Belkhale and Banerjee [1] have studied the problem using a speedup function which is no greater than linear. In this paper we are concerned with the problem where precedence relations exist among parallel tasks, and we use the linear speedup assumption.

The justification of defining the maximum degree of parallelism for each task is that in many operations, complete parallelism is difficult to achieve, but limited parallelism may be accomplished easily. At the same time, a linear reduction in execution time is a reasonable assumption if only simple parallelizations were attempted on tasks. Our problem is NP-hard since the NP-hard problem of nonpreemptive scheduling of sequential tasks [7] is a special case of our problem when the degree of parallelism is limited to 1 for every task. Since polynomial time optimal solution is unlikely for any NP-hard problem, we are interested in polynomial time approximation algorithms. A heuristic scheduling algorithm H has a performance bound α if for all problem instances, it can guarantee that

$$\frac{M_H(\mathcal{T},\mathcal{P})}{M_{OPT}(\mathcal{T},\mathcal{P})} \leq \alpha, \tag{1.2}$$

where $M_{OPT}(\mathcal{T},\mathcal{P})$ and $M_H(\mathcal{T},\mathcal{P})$ are the optimal and the heuristic makespans, respectively. When the bound is tight, algorithm H is said to have the performance ratio α. Recently, we have proved in [11] that a list scheduling algorithm which uses the earliest completion time heuristic achieves a performance bound of $\ln \Delta + 2$. In this paper we improve the performance bound of that algorithm to 3 and give an example to show that the bound is at least 2.5.

The rest of the paper is organized as follows. In §2 the original list scheduling algorithm [7] is presented together with its performance. The earliest completion time algorithm and its analysis are presented in §§3–5. In §6 an example is presented to demonstrate that the bound is at least 2.5. Concluding remarks are given in §7.

2. The original list scheduling algorithm. The list scheduling algorithm LS proposed by Graham [7] for sequential tasks is formally presented in Algorithm 1. In the algorithm, t_{next} is basically the earliest time at which there is at least one free processor. Initially $t_{\text{next}} = 0$. At time t_{next}, the algorithm selects the first ready task from the given task list for execution. If there is no ready task at time t_{next}, t_{next} is then assigned

the earliest time at which at least one more running task completes its execution. The algorithm repeats these simple steps until the task list becomes empty.

---

input(task list $\mathcal{T}$)

$t_{\text{next}} \leftarrow 0$

repeat

  $t_{\text{next}} \leftarrow \max\{t_{\text{next}}$, time when at least one processor is free$\}$

  if (there exists a ready task in the list $\mathcal{T}$ at t_{next}) **then**

    remove the first ready task T_j from the list $\mathcal{T}$

(†)   execute T_j on processor P_i where P_i is free at t_{next}

  else

    $t_{\text{next}} \leftarrow$ time with at least one more task completes

  endif

until $\mathcal{T} = \emptyset$

ALGORITHM 1. *The original list scheduling algorithm.*

Applying algorithm LS to a list of parallel tasks $\mathcal{T}$ is essentially giving each task a scheduled parallelism of 1, and it may cause underutilization of the processor system. Let $[t_{2i-1}, t_{2i})$, $i = 1, \cdots, k$ be the time intervals of the LS schedule having idle processors. Graham [7] has shown that there must be a chain of tasks in $\mathcal{T}$,

$$T_{j_h} \prec \cdots \prec T_{j_i} \prec \cdots \prec T_{j_1}, \tag{2.1}$$

such that $\cup_{i=1}^{k}[t_{2i-1}, t_{2i}) \subseteq \cup_{i=1}^{h}[b_{LS}(T_{j_i}), c_{LS}(T_{j_i}))$. Based on this observation, Wang and Cheng [11] have shown that the performance ratio of algorithm LS is $\frac{M_{LS}(\mathcal{T},\mathcal{P})}{M_{OPT}(\mathcal{T},\mathcal{P})} \leq \Delta + \frac{m-\Delta}{m}$, where Δ is the maximum degree of parallelism in any task. Note that when $\Delta = 1$, the bound is reduced to that obtained in [7] for scheduling sequential tasks.

3. The ECT scheduling algorithm. The ECT algorithm is basically a list scheduling algorithm, except that the (†) statement in Algorithm 1 is replaced by an algorithm to determine the execution time and the scheduled parallelism of a ready task. The idea of the algorithm is to complete the ready task as soon as possible, and it is called the *Earliest Completion Time* (ECT) algorithm (Algorithm 2). For each parallel task scheduled, the time at which the task T_j was submitted to the ECT algorithm is called its *preparation* time, denoted as $a_{ECT}(T_j)$. In the rest of the paper, the subscript *ECT* will be omitted. Obviously,

$$r(T_j) \leq a(T_j) \leq b(T_j) \leq c(T_j) \quad \forall T_j \in \mathcal{T}. \tag{3.1}$$

We say that the task T_j is being *prepared* from time $a(T_j)$ to $b(T_j)$.

To visualize processor allocation, for each processor $P_i \in \mathcal{P}$ we present its activities by using a half-open strip starting from time $t = 0$. The activities of all processors in $\mathcal{P}$ may be expressed by m stacked-up strips, P_i, $i = 1, \cdots, m$. Initially, all strips are

---

procedure *Earliest_Completion* (t_{next}, T_j)

$p' \leftarrow 1;\ t_{p'} \leftarrow t_{\text{next}};\ t_{\text{earliest}} \leftarrow t_{\text{next}} + \tau_j$

for $p \leftarrow 2$ **to** δ_j **do**

  find the earliest time $t_p \geq t_{\text{next}}$ with p free processors

  $\tilde{t}_p \leftarrow t_p + \tau_j/p$

  if $\tilde{t}_p < t_{\text{earliest}}$ **then** $t_{\text{earliest}} \leftarrow \tilde{t}_p;\ p' \leftarrow p;\ t_{p'} \leftarrow t_p$

enfor

schedule T_j on p' free processors at time $t_{p'}$; $t_{\text{next}} \leftarrow t_{p'}$

ALGORITHM 2. *The Earliest Completion Time algorithm.*

white (not colored), indicating that there is no activity. Since all processors are identical, we may assume without loss of generality that at time t_p, the available processors are $P_1, \cdots, P_p$ where $p \in \{1, \cdots, \delta_j\}$. Figure 1(a) illustrates the fitting process by the ECT algorithm for a task with $\delta_j = 5$ and $p' = 3$. For the strip (processor) P_p, $1 \leq p \leq p'$, we color it *grey* from the time t_p to the task's start time $b(T_j)$. For the p' strips on which the task was scheduled, they are colored *black* from the time $b(T_j) = t_{p'}$ to $t_{p'} + \tau_j/p'$. The coloring of the task in Fig. 1(a) is a black rectangle with a staircase-shaped grey area to its left (Fig. 1(b)). Note that it is impossible to recolor any area that is already black or grey. It is easy to see that a necessary and sufficient condition for a section of a strip to remain white is when the corresponding processor is free but no ready task is available. An interval $[t', t'')$ with $t'' \leq M_{ECT}(\mathcal{T}, \mathcal{P})$ is incompletely colored if for all t, $t \in [t', t'')$, there is at least one strip whose color is white at time t; otherwise, it is completely colored. We further assume that each such interval is of maximal length, i.e., it is not contained in another incompletely colored interval. Let k be the number of incompletely colored intervals in the ECT schedule. These intervals may be represented by

$$[t_1, t_2), \cdots, [t_{2k-1}, t_{2k}). \tag{3.2}$$

LEMMA 1. *In the* ECT *schedule,* $[r(T_j), b(T_j)) \cap \{\cup_{i=1}^{k}[t_{2i-1}, t_{2i})\} = \emptyset$ *for any task* T_j.

Proof. The case where $r(T_j) = b(T_j)$ is obvious since $[\alpha, \alpha) = \emptyset$ for any number α. Hence, we may assume without loss of generality that $r(T_j) < b(T_j)$. It is sufficient to

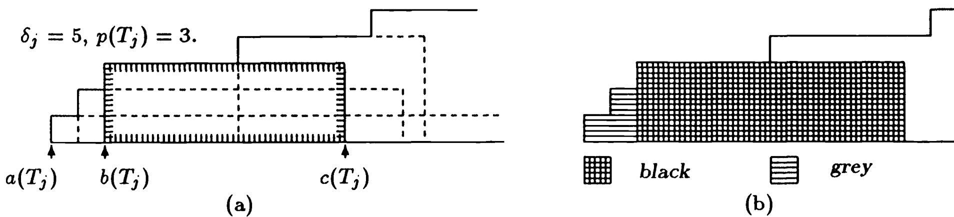


FIG. 1. *Fitting and coloring in earliest completion time algorithm.*

show that the interval $[r(T_j), b(T_j))$ is completely colored. Since the task T_j was ready but was not prepared before $a(T_j)$, each strip must be completely colored (black or grey) in the interval $[r(T_j), a(T_j))$. From $a(T_j)$ to $b(T_j)$, each strip is either colored black by other scheduled tasks or colored grey by the preparation of T_j. By definition, completely and incompletely colored intervals cannot overlap one another. Therefore, the lemma is true. □

Let $[t_{2i-1}, t_{2i}), i = 1, \cdots, k$ be the incompletely colored intervals of the ECT schedule. Similar to the proof of the LS schedule, the next lemma establishes the fact that the incompletely colored intervals in any ECT schedule must be covered by a chain of tasks.

LEMMA 2 (Covering lemma). *There must be a chain of tasks in $\mathcal{T}$,*

$$T_{j_h} \prec \cdots \prec T_{j_i} \prec \cdots \prec T_{j_1}, \tag{3.3}$$

such that $\cup_{i=1}^{k}[t_{2i-1}, t_{2i}) \subseteq \cup_{i=1}^{h}[b(T_{j_i}), c(T_{j_i}))$.

Proof. We first choose the task T_{j_1} that finishes last in the schedule, i.e., $c(T_{j_1}) = \max_{T_j \in \mathcal{T}} c(T_j)$. If $r(T_{j_1}) = 0$, the execution interval of T_{j_1}, $[b(T_{j_1}), c(T_{j_1}))$ covers all incompletely colored intervals. This is because by Lemma 1, the interval $[r(T_{j_1}), b(T_{j_1}))$ overlaps no incompletely colored interval. If T_{j_1} is not ready at time $t = 0$, we can find its predecessor task T_{j_2} such that $c(T_{j_2}) = r(T_{j_1})$. In general, if the task T_{j_i} is not ready at time $t = 0$, we can find its predecessor $T_{j_{i+1}}$ such that $c(T_{j_{i+1}}) = r(T_{j_i})$. After a finite number of steps, we will eventually find a task T_{j_h} ready at $t = 0$. The selected sequence of tasks forms the chain of (3.3). By Lemma 1, no incompletely colored interval can overlap $[c(T_{j_{i+1}}), b(T_{j_i}))$, $i = 1, \cdots, h-1$, nor can any incompletely colored interval overlap the interval $[0, b(T_{j_h}))$. Therefore, the incompletely colored intervals must be completely covered by the tasks in the chain of (3.3). □

4. The sequences of wide tasks. To analyze the size of the black areas, we examine closely how tasks were prepared and scheduled. A task is called a *wide* task if its maximum degree of parallelism is at least $\lceil m/2 \rceil$; otherwise, it is called a *narrow* task. In other words, $\mathcal{T}$ is divided into the subsets of wide and narrow tasks, $\mathcal{T} = \mathcal{T}_W \cup \mathcal{T}_N$. First we will prove that in the intervals covered by the preparation and the execution of wide tasks, the black area is at least half the total area. Then for all the remaining intervals not covered by the wide tasks, they are further divided into two groups of subintervals, H and I. We will show that at least half of the area in H is colored black, and I is covered by the shortest possible execution time of a subchain from the task chain of (3.3). Using these results, we establish the $3 - \frac{2}{m}$ performance bound of the ECT algorithm.

Let T_j be a wide task. A wide task T_h is an *immediate follower* of task T_j if it satisfies

$$c(T_h) = \max_{a(T_j) < a(T_{j'}) \leq c(T_j),\ T_{j'} \in \mathcal{T}_W} c(T_{j'}) \quad \text{and} \quad c(T_j) < c(T_h). \tag{4.1}$$

Similarly, a wide task T_i is a *distant follower* of task T_j if it satisfies

$$c(T_i) = \max_{a(T_{j'}) = t,\ T_{j'} \in \mathcal{T}_W} c(T_{j'}), \quad \text{where } t = \min_{c(T_j) < a(T_{j'}),\ T_{j'} \in \mathcal{T}_W} a(T_{j'}). \tag{4.2}$$

To construct sequences of wide tasks, the first wide task $T_{(1,1)}$ is selected to satisfy

$$c(T_{(1,1)}) = \max_{a(T_{j'}) = t,\ T_{j'} \in \mathcal{T}_W} c(T_{j'}), \quad \text{where } t = \min_{0 \leq a(T_{j'}),\ T_{j'} \in \mathcal{T}_W} a(T_{j'}). \tag{4.3}$$

In general, suppose we have already found the wide task $T_{(i,j)}$. If $T_{(i,j)}$ has an immediate follower, we will designate that follower as $T_{(i,j+1)}$. If it does not have an immediate follower, but has a distant follower, we will designate the distant follower as $T_{(i+1,1)}$. The selection is repeated until we can find neither an immediate follower nor a distant follower. The wide tasks selected in this process are shown as follows:

$$\begin{array}{ccc} T_{(1,1)}, & \cdots, & T_{(1,j_1)} \\ \vdots & \cdots & \\ T_{(l,1)}, & \cdots, & T_{(l,j_l)}. \end{array} \tag{4.4}$$

LEMMA 3. *There is no wide task that is either prepared or executed in the following intervals:* $[0, a(T_{(1,1)})), \cdots, [c(T_{(i,j_i)}), a(T_{(i+1,1)})), \cdots, [c(T_{(l,j_l)}), M_{ECT}(\mathcal{T}, \mathcal{P}))$.

Proof. If the lemma is not true, we can assume that there is a wide task T' with

$$c(T_{(i,j_i)}) \leq a(T') < a(T_{(i+1,1)}) \quad \text{or} \quad a(T') \leq c(T_{(i,j_i)}) < c(T'). \tag{4.5}$$

Without loss of generality, we can further assume that T' has the latest completion time $c(T')$ among all tasks satisfying the conditions in (4.5).

If $c(T_{(i,j_i)}) \leq a(T') < a(T_{(i+1,1)})$, the inclusion of $T_{(i+1,1)}$ in (4.4) contradicts the rule of selecting $T_{(i,j_i)}$'s distant follower because T' or some wide task before it should be selected.

If $a(T') \leq c(T_{(i,j_i)}) < c(T')$, then there are two cases:

(a) $a(T_{(i,j_i)}) < a(T')$. The task $T_{(i,j_i)}$ should not be the last task in the sequence i because T' is its immediate follower.

(b) $a(T_{(i,j_i)}) \geq a(T')$. By the definition of an immediate follower, the task T' should be chosen instead of the task $T_{(i,j_i)}$ because the task T' has a later task completion time.

The contradictions in all possible cases have proved the lemma. $\square$

Next we divide the interval covered by a single row of wide tasks

$$T_{(i,1)}, \cdots, T_{(i,j_i)} \tag{4.6}$$

into subintervals: $[t_{h-1}, t_h)$, $h = 1, \cdots, j_i$, where $t_0 = a(T_{(i,1)})$, and $t_h = c(T_{(i,h)})$. Suppose we remove the black areas associated with the tasks of (4.6) in the ECT schedule, and rearrange the remaining black strips in each individual subintervals into staircase-shaped regions in such a way that the size of the remaining black-colored regions is unchanged (Fig. 2). Within the subinterval $[t_{h-1}, t_h)$, we now try to fit into the area at the lower right-hand corner a black rectangle which is the largest possible in area and with height not exceeding $\delta_{(i,h)}$. Figure 3 shows such a fitting for some subintervals.

LEMMA 4. *For each subinterval* $[t_{h-1}, t_h)$, $1 \leq h \leq j_i$, *the area of the largest black rectangle fitted will not exceed* $\tau_{(i,h)}$.

Proof. If this is not true, we can schedule the task $T_{(i,h)}$ differently and achieve a completion time earlier than $t_h = c(T_{(i,h)})$. This is in contradiction with the scheduling criterion used in the ECT algorithm. $\square$

Now we estimate the total size of all black areas in each subinterval $[t_{h-1}, t_h)$. Let R_h and Q_h be the size of the rectangle at the lower right-hand corner and the staircase-shaped region at the upper left-hand corner, respectively. Furthermore, let L be the length of the interval, i.e., $L = t_h - t_{h-1}$, and δ the maximum number of strips (processors) that task $T_{(i,h)}$ can possibly use, i.e., $\delta = \delta_{(i,h)} = \delta(T_{(i,h)})$.

LEMMA 5. *For any positive integer* $\delta \geq \lceil m/2 \rceil$, $R_h + Q_h \geq \frac{m \cdot L}{2}$.

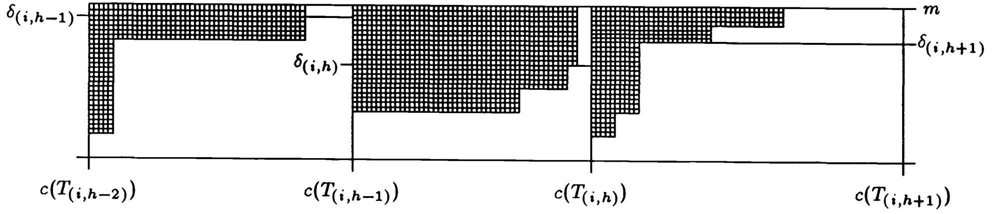


FIG. 2. *Black-colored regions after rearrangement.*

Proof. To make the mathematical analysis simpler, we analyze the mirror reflection of the region (Fig. 4(a)). To estimate Q_h, we compute Q'_h (Fig. 4(b)), which is the area of the convex region bounded by $xy = R_h$, $x = L$, $y = m$, and $x = v$ where (v, δ) is the intersection point of the curve $xy = R_h$ with the horizontal line $y = \delta$. It is not difficult to see that the region with area Q'_h is completely inside the region with area Q_h; otherwise a rectangle with area larger than R_h is possible. Hence $Q'_h < Q_h$. Since the curve in Fig. 4(b) represents the equation $xy = R_h$, the area of the black regions in Fig. 4(b) is exactly the same as those in Fig. 5.

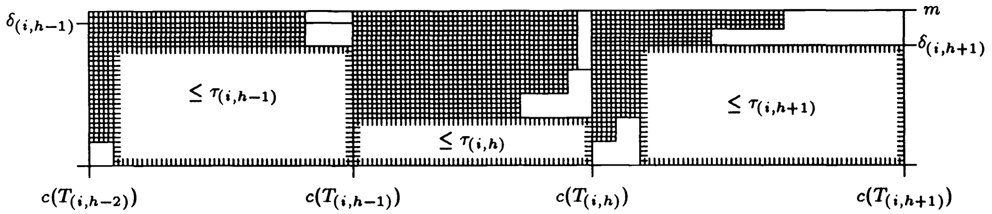


FIG. 3. *Fitting of largest rectangles.*

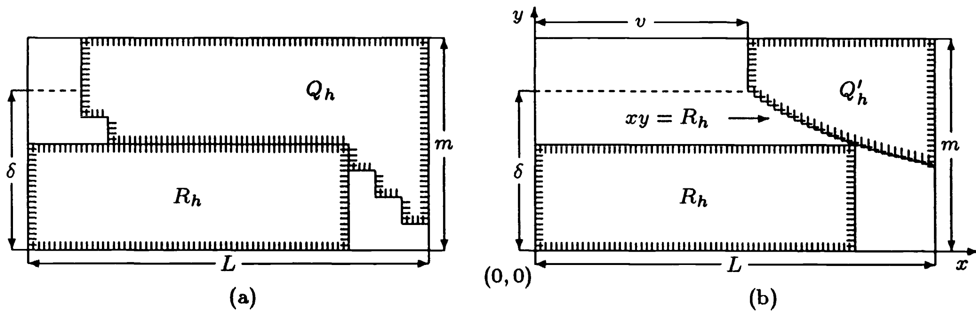


FIG. 4. *Estimating the black-colored areas.*

Let (L, u) be the intersection point of the curve $xy = R_h$ with the vertical line $x = L$. Then

$$R_h = \delta \cdot v = u \cdot L. \tag{4.7}$$

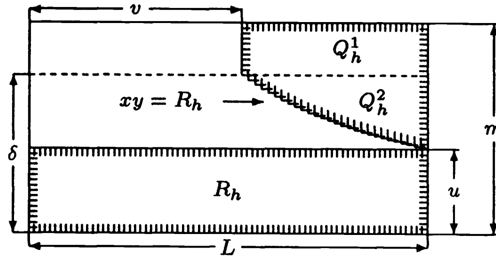


FIG. 5. *Analysis of the black-colored areas.*

Now

$$(4.8)\qquad \begin{aligned} Q_h^1 &= (m-\delta)(L-v),\\ Q_h^2 &= \delta L - uL - \textstyle\int_u^\delta R_h \cdot dy/y,\\ Q_h' &= Q_h^1 + Q_h^2 = mL - muL/\delta - uL\ln\delta + uL\ln u. \end{aligned}$$

Hence

$$(4.9)\qquad R_h + Q_h > R_h + Q_h' = uL + mL - muL/\delta - uL\ln\delta + uL\ln u.$$

Let $f(u,\delta) = R_h + Q_h'$ where $m/2 \le \delta \le m$ and $1 \le u \le \delta$. By simple calculus,

$$\partial f/\partial u = 2L - mL/\delta - L\ln\delta + L\ln u,$$
$$\partial f/\partial u = 0 \Leftrightarrow \ln u = m/\delta + \ln\delta - 2 \Leftrightarrow u = \delta \cdot e^{m/\delta - 2}.$$

Since

$$\frac{\partial^2 f}{\partial u^2} = \frac{L}{u} > 0 \quad \forall u \in [1,\delta],$$

for any fixed δ, $f(u,\delta)$ reaches its minimum at $u = \delta e^{m/\delta-2}$. Let $g(\delta) = f(\delta e^{m/\delta-2},\delta)$, then

$$g(\delta) = mL - \delta e^{m/\delta-2}L,$$
$$g'(\delta) = (me^{m/\delta-2}/\delta - e^{m/\delta-2})L = 0 \Leftrightarrow m/\delta - 1 = 0 \Leftrightarrow m = \delta.$$

It is not difficult to check that $g(\delta)$ reaches a maximum at $\delta = m$, so for $m/2 \le \delta \le m$, g has a minimum value of $mL/2$ at $\delta = m/2$, and so does the function f, i.e.,

$$(4.10)\qquad f(u,\delta) \ge \frac{m\cdot L}{2} \quad \forall u \in [1,\delta] \text{ and } \forall \delta \in \left[\frac{m}{2}, m\right].$$

Combining with the inequality (4.9), the inequality in the lemma must be true. □

By Lemmas 4 and 5, we can conclude that the total black area occurring between t_0 and t_{j_i} is at least $m(t_{j_i} - t_0)/2$. Let W_i and W_i^b be the available and the utilized processing power in the ith wide task sequence in (4.4), respectively. It is obvious that Lemma 6 follows by applying the result to all l wide task sequences.

LEMMA 6.

$$(4.11)\qquad \sum_{i=1}^{l} W_i^b \ge \frac{1}{2}\sum_{i=1}^{l} W_i.$$ □

5. Overall analysis of the performance bound. Now we investigate the intervals that were not covered by the preparation and the execution of wide tasks' sequences in (4.4). For W_i, $i = 1, \cdots, l$, let $[t^{(2i-1)}, t^{(2i)})$ be its corresponding interval on the x-axis. We further use W to denote the union of all such intervals, i.e., $W = \cup_{i=1}^{l}[t^{(2i-1)}, t^{(2i)})$. Excluding the intervals in W, we have $l+1$ intervals that are not covered by the l sequences of wide tasks:

$$(5.1) \qquad [t^{(0)}, t^{(1)}), [t^{(2)}, t^{(3)}), \cdots, [t^{(2i)}, t^{(2i+1)}), \cdots, [t^{(2l)}, t^{(2l+1)}),$$

where $t^{(0)} = 0$ and $t^{(2l+1)} = M_{ECT}(\mathcal{T}, \mathcal{P})$. If the first wide task sequence starts at $t^{(0)}$, or the last wide task sequence terminates at $M_{ECT}(\mathcal{T}, \mathcal{P})$, we can simply have $t^{(0)} = t^{(1)}$ or $t^{(2l)} = t^{(2l+1)}$.

LEMMA 7. *In an* ECT *schedule, we cannot have grey and white strips coexist at any time* $t \in [0, M_{ECT}(\mathcal{T}, \mathcal{P}))$.

Proof. From the ECT algorithm, we know that the grey color is used during the preparation but before the execution of a task T_j. If at least one processor (strip) is colored grey at time t, all other free processors must also be colored grey because the earliest completion time scheduling policy colors all white strips grey during the preparation of any ready task. Therefore, grey and white colors cannot coexist at any time in the schedule. □

We further subdivide the intervals in (5.1) into *more than half-colored* and *at most half-colored* categories. An interval is more than half colored if at all time in the interval, the number of white strips is less than $m/2$. On the other hand, in an at most half-colored interval, the number of white strips is greater than or equal to $m/2$ at all time t. For each time interval $[t^{(2i)}, t^{(2i+1)})$, $i = 0, \cdots, l$, let

$$(5.2) \qquad [t_1^{(2i)}, t_2^{(2i)}), \cdots, [t_{2j-1}^{(2i)}, t_{2j}^{(2i)}), \cdots, [t_{2k_i-1}^{(2i)}, t_{2k_i}^{(2i)})$$

be its more than half-colored intervals. We use H to denote the union of all such subintervals, i.e., $H = \cup_{i=0}^{l} \cup_{j=1}^{k_i} [t_{2j-1}^{(2i)}, t_{2j}^{(2i)})$.

LEMMA 8. *For any more than half-colored interval* $[t_{2j-1}^{(2i)}, t_{2j}^{(2i)})$, *the rectangle* $m \times (t_{2j}^{(2i)} - t_{2j-1}^{(2i)})$ *has more than half of its area colored black.*

Proof. Based on Lemma 7, we can consider the following two cases.

1. All strips in $[t_{2j-1}^{(2i)}, t_{2j}^{(2i)})$ are colored black and white: the statement in the lemma is true based directly on the definition of the more than half-colored interval.

2. All strips in $[t_{2j-1}^{(2i)}, t_{2j}^{(2i)})$ are colored black and grey: we can conclude from Lemma 3 that the grey area in $[t_{2j-1}^{(2i)}, t_{2j}^{(2i)})$ must be introduced during the preparation of a narrow task. At any moment of preparing a narrow task in the ECT algorithm, less than $\lfloor m/2 \rfloor$ strips are colored grey. Therefore, more than half of the total rectangular area is colored black. □

LEMMA 9. *If a task* T_j *is scheduled to start execution in an incompletely colored interval* $[t', t'')$, *the task* T_j *must be scheduled to run at its maximum degree of parallelism, i.e.,*

$$b(T_j) \in [t', t'') \Rightarrow p(T_j) = \delta_j.$$

Proof. Although we are only interested in the narrow tasks, this lemma is true for all tasks, narrow or wide. If a task T_j is scheduled to start execution in an incompletely colored interval, we must have more processors than are really needed at time $b(T_j)$. From the way tasks were scheduled by the ECT algorithm, the task T_j must have been

scheduled to its maximum degree of parallelism; otherwise, we can further shorten the execution time of T_j, which contradicts the scheduling criterion used in the ECT algorithm. □

LEMMA 10. *For an at most half-colored interval* $[t_{2j}^{(2i)}, t_{2j+1}^{(2i)})$ *and a narrow task* T_k,

$$t_{2j}^{(2i)} \in [b(T_k), c(T_k)) \Rightarrow c(T_k) - t_{2j}^{(2i)} \leq \frac{\tau_k}{\delta_k}. \tag{5.3}$$

Proof. If the task under investigation T_k has already been executed with its maximum degree of parallelism, the statement (5.3) is obviously true.

Let $\tilde{t} = b(T_k)$ be the scheduled time for T_k, and let $\tilde{p}$ be the number of processors for T_k's execution. There is also a $t_{\delta_k} \geq \tilde{t}$ which is the earliest time task T_k can be parallelized to the maximum extent. We must also have $t_{\delta_k} \leq t_{2j}^{(2i)}$ because at time $t_{2j}^{(2i)}$ there are at least $m/2$ free processors, which are more than enough to fully parallelize the narrow task T_k. From the way the execution time of a task is determined in the ECT algorithm, we have $c(T_k) \leq t_{\delta_k} + \tau_k/\delta_k$. Therefore,

$$c(T_k) - t_{2j}^{(2i)} \leq c(T_k) - t_{\delta_k} \leq \frac{\tau_k}{\delta_k}. \qquad \square$$

Let I be the union of all at most half-colored intervals that are not yet covered by the wide tasks, i.e.,

$$I = [0, M_{ECT}(\mathcal{T}, \mathcal{P})) - W - H. \tag{5.4}$$

From the task chain T_{j_i}, $i = 1, \cdots, h$, constructed in Lemma 2, there is a subchain of narrow tasks

$$T_{j'_g} \prec \cdots \prec T_{j'_i} \prec \cdots \prec T_{j'_1}, \tag{5.5}$$

which not only covers the intervals in I, it also provides an upper bound on the total length of these intervals. Lemma 11 shows that the total length of the intervals in I does not exceed the execution time of the narrow task chain (5.5) even when it is executed with maximum parallelism.

LEMMA 11. $\sum_{i=1}^{g} \tau_{j'_i}/\delta_{j'_i} \geq |I|$, *where* $|I|$ *is the total length of intervals in* I.

Proof. The intervals in I is a subset of all incompletely colored intervals, so we must have a subchain from (3.3) to cover them. From Lemma 3, the intervals that are not covered by (5.1) must be covered by narrow tasks. Hence, we have a subchain of narrow tasks that covers I. Now for each task $T_{j'_i}$ in the subchain, there are two cases.

(1) If $p(T_{j'_i}) = \delta_{j'_i}$, then the section(s) of $T_{j'_i}$ overlapping I sum in length to at most $\tau_{j'_i}/\delta_{j'_i}$.

(2) If $p(T_{j'_i}) < \delta_{j'_i}$, then by Lemma 9, $b(T_{j'_i}) < t'$ for any interval $[t', t'')$ of I that overlaps $T_{j'_i}$. In particular, let $[t', t'')$ be the earliest interval of I that overlaps $T_{j'_i}$. By Lemma 10, $c(T_{j'_i}) - t' \leq \tau_{j'_i}/\delta_{j'_i}$, from which we also conclude that the section(s) of $T_{j'_i}$ overlapping I sum in length to at most $\tau_{j'_i}/\delta_{j'_i}$.

Combining these two cases yields the lemma. □

Let us use $H_{(i,j)}$ to denote the rectangle of size $m \times (t_{2j}^{(2i)} - t_{2j-1}^{(2i)})$, and whose bottom edge is the interval $[t_{2j-1}^{(2i)}, t_{2j}^{(2i)})$. According to Lemma 8, at least one-half of the area in $H_{(i,j)}$ is black. In other words, if $H_{(i,j)}^b$ denotes the black area in $H_{(i,j)}$, we must have

$$\sum_{i=0}^{l} \sum_{j=1}^{k_i} H_{(i,j)}^b \geq \frac{1}{2} \sum_{i=0}^{l} \sum_{j=1}^{k_i} H_{(i,j)}. \tag{5.6}$$

THEOREM 1. *The overall performance bound of the* ECT *schedule is* $3 - 2/m$, *i.e.*,

$$\frac{M_{ECT}(\mathcal{T},\mathcal{P})}{M_{OPT}(\mathcal{T},\mathcal{P})} \leq 3 - \frac{2}{m}. \tag{5.7}$$

Proof. We first show that

$$M_{OPT}(\mathcal{T},\mathcal{P}) \geq \sum_{i=1}^{g} \frac{\tau_{j'_i}}{\delta_{j'_i}} \geq |I|. \tag{5.8}$$

The task subchain (5.5) satisfies the precedence relationship, so the optimal makespan cannot be shorter than the total execution time of the tasks in (5.5) even if each task is executed with maximum parallelism, i.e.,

$$M_{OPT}(\mathcal{T},\mathcal{P}) \geq \sum_{i=1}^{g} \frac{\tau_{j'_i}}{\delta_{j'_i}}.$$

Combining with the inequality in Lemma 11, we have (5.8).

Let $|H|$ and $|W|$ denote the total length of more than half-colored intervals in (5.2) and intervals covered by wide tasks in (4.4), respectively. We next show that

$$M_{OPT}(\mathcal{T},\mathcal{P}) \geq \frac{1}{m}|I| + \frac{1}{2m}\left(\sum_{i=0}^{l}\sum_{j=1}^{k_i} H_{(i,j)} + \sum_{i=1}^{l} W_i\right) = \frac{1}{m}|I| + \frac{1}{2}(|H| + |W|). \tag{5.9}$$

The black area in either $H_{(i,j)}$ or W_i represents the utilized processing power during the execution of the tasks in $\mathcal{T}$, and in the intervals of I, there is at least one utilized processor. Therefore, we must have

$$M_{OPT}(\mathcal{T},\mathcal{P}) \geq \frac{1}{m}|I| + \frac{1}{m}\left(\sum_{i=0}^{l}\sum_{j=1}^{k_i} H^b_{(i,j)} + \sum_{i=1}^{l} W^b_i\right). \tag{5.10}$$

Combining with the two inequalities in Lemma 6 and (5.6), we have (5.9).

From (5.8) and (5.9), we have

$$M_{ECT}(\mathcal{T},\mathcal{P}) = |I| + |H| + |W| \leq \left(3 - \frac{2}{m}\right) M_{OPT}(\mathcal{T},\mathcal{P}). \tag{5.11}$$

□

6. Tightness of the bound. In this section we will construct an example to show that

$$2.5 \leq \sup\left\{\frac{M_{ECT}(\mathcal{T},\mathcal{P})}{M_{OPT}(\mathcal{T},\mathcal{P})}\right\} \leq 3 - \frac{2}{m}. \tag{6.1}$$

There are $n = m - \lfloor m/e \rfloor + 2$ tasks in $\mathcal{T}$, where $m = |\mathcal{P}|$ and e is the base of the natural logarithmic function. The size and the maximum degree of parallelism are given as follows:

$$\begin{aligned} \tau_1 &= L - L/e, & \delta_1 &= 1, \\ \tau_j &= L + \epsilon - (m/e)L/(m+1-j), & \delta_j &= 1, \quad j = 2, \cdots, n-2, \\ \tau_{n-1} &= mL/e, & \delta_{n-1} &= m, \\ \tau_n &= 2L/3, & \delta_n &= 1, \end{aligned} \tag{6.2}$$

where L is a constant, and ϵ is an arbitrarily small number such that $\epsilon \leq L/(e(m-1))$. We also assume that the precedence relationship is empty among all tasks. It is easy to see that the ECT schedule of $\mathcal{T}$ may be graphically represented in Fig. 6(a). The use of ϵ is to ensure that the task T_{n-1} is scheduled to fully utilize all processors in the system by the ECT algorithm. Otherwise, if ϵ is absent, the task T_{n-1} will use $\lfloor m/e \rfloor + 1$ processors and still finish at time L. The makespan of the ECT schedule is

$$M_{ECT}(\mathcal{T}, \mathcal{P}) = \frac{5}{3}L. \tag{6.3}$$

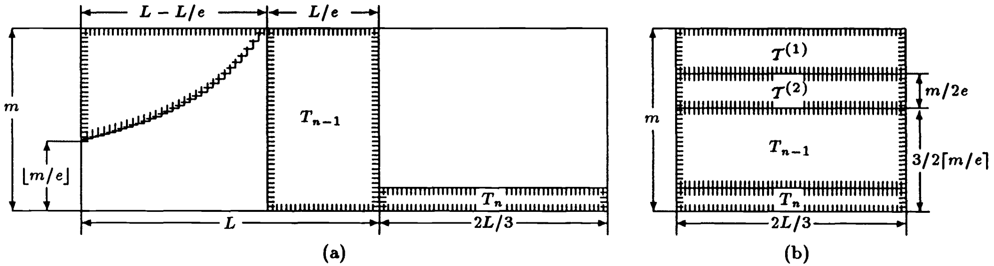


FIG. 6. *An example with performance ratio* 2.5.

To derive the optimal schedule of $\mathcal{T}$, let $n' = m - 2\lceil m/e \rceil - 1$, and the first $n-2$ tasks in $\mathcal{T}$ may be separated into two groups,

$$\begin{aligned} \mathcal{T}^{(1)} &= \{T_1, \cdots, T_{n'}\}, \\ \mathcal{T}^{(2)} &= \{T_{n'+1}, \cdots, T_{n-2}\}. \end{aligned} \tag{6.4}$$

In the second group $\mathcal{T}^{(2)}$, the tasks are put into pairs in the following way:

$$T_{p(i)} = \{T_{i'}, T_{i''}\}, \qquad i' = m - 2\left\lceil \frac{m}{e} \right\rceil + i - 1, \quad i'' = m - \left\lceil \frac{m}{e} \right\rceil - i + 2, \tag{6.5}$$

with $i = 1, \cdots, (1/2)\lceil m/e \rceil + 1$. Assume without loss of generality that there are an even number of tasks in $\mathcal{T}^{(2)}$, so all of them can be successfully paired.

In the optimal schedule of $\mathcal{T}$, we execute all tasks in $\mathcal{T}^{(1)}$ at time $t = 0$. For every pair of tasks in $\mathcal{T}^{(2)}$, we execute $T_{i'}$ at time $t = 0$, and $T_{i''}$ immediately after $T_{i'}$. The task T_n should also be executed at $t = 0$. Since each of these tasks uses only one processor, the only remaining task T_{n-1} will have $(3/2)\lceil m/e \rceil - 1$ processors left for its execution at time $t = 0$. The graphic representation of the optimal schedule is approximated in Fig. 6(b). We now show that all these tasks can be completed no later than $M_{OPT}(\mathcal{T}, \mathcal{P}) = 2L/3 + o(1)$ where $o(1)$ is a quantity that can be made arbitrarily small.

Since the size of the task T_n is $2L/3$, it can be completed in time $2L/3$. To prove that all tasks in $\mathcal{T}^{(1)}$ are completed no later than $2L/3 + o(1)$, we observe the following inequality:

$$\tau_j \leq \left(1 - \frac{1}{e}\right) L + o(1) < \frac{2L}{3} + o(1) \quad \forall T_j \in \mathcal{T}^{(1)}. \tag{6.6}$$

For the task T_{n-1}, since we execute it with $(3/2)\lceil m/e \rceil - 1$ processors starting from time $t = 0$, its completion time is

$$\frac{\tau_{n-1}}{\frac{3}{2}\lceil \frac{m}{e} \rceil - 1} < \frac{\frac{m}{e}L}{\frac{3m}{2e} - 1} = \frac{2}{3}L + o(1). \tag{6.7}$$

Finally, we show that any pair of tasks in $\mathcal{T}^{(2)}$ will be completed no later than $2L/3+o(1)$. For any such pair of tasks in $\mathcal{T}^{(2)}$, since they are executed on a single processor one after the other, it is sufficient to prove the following equivalent statement:

$$\tau_{i'} + \tau_{i''} \leq \frac{2}{3}L + o(1). \tag{6.8}$$

We have

$$\begin{aligned} \tau_{i'} + \tau_{i''} &= L - (\tfrac{m}{e})L/(m+1-i') + L - (\tfrac{m}{e})L/(m+1-i'') + 2\epsilon \\ &= 2L - (\tfrac{m}{e})L/(2\lceil \tfrac{m}{e} \rceil - i + 2) - (\tfrac{m}{e})L/(\lceil \tfrac{m}{e} \rceil + i - 1) + 2\epsilon, \end{aligned} \tag{6.9}$$

with $i = 1, \cdots, (1/2)\lceil m/e \rceil + 1$. To estimate the upper bound of (6.9), we define the following function:

$$h(z) = 2L - \frac{\frac{m}{e}L}{2\frac{m}{e} - z} - \frac{\frac{m}{e}L}{\frac{m}{e} + z}, \quad \text{with } z \in \left[0, \frac{m}{2e}\right]. \tag{6.10}$$

It is quite obvious that

$$\tau_{i'} + \tau_{i''} \leq \sup\{h(z)\} + o(1) \quad \forall i', i''. \tag{6.11}$$

We now calculate the first derivative of $h(z)$,

$$h'(z) = \frac{-\frac{m}{e}L}{(2\frac{m}{e} - z)^2} + \frac{\frac{m}{e}L}{(\frac{m}{e} + z)^2}. \tag{6.12}$$

We then have the following:

$$h'(z) = 0 \Rightarrow \left(2\frac{m}{e} - z\right)^2 = \left(\frac{m}{e} + z\right)^2 \Rightarrow z = \frac{m}{2e}. \tag{6.13}$$

Comparing functional values of $h(z)$ at zero and $m/2e$, we can conclude that

$$\sup\{h(z)\} = h\left(\frac{m}{2e}\right) = 2L - \frac{4L}{3} = \frac{2L}{3}. \tag{6.14}$$

Hence the performance ratio of the ECT schedule with respect to the optimal schedule is

$$\frac{M_{ECT}(\mathcal{T}, \mathcal{P})}{M_{OPT}(\mathcal{T}, \mathcal{P})} = \frac{\frac{5}{3}L}{\frac{2}{3}L + o(1)}, \tag{6.15}$$

which approaches 2.5 asymptotically. Therefore, the worst-case performance ratio of the ECT algorithm falls between 2.5 and $3 - 2/m$ as given in (6.1).

7. Conclusion. In this paper we have investigated the ECT algorithm for scheduling parallel tasks, and showed that the worst case ratio of the ECT schedule to the optimal schedule falls between 2.5 and $3 - 2/m$. Further research on this problem may concentrate on closing the gap between these two values.

REFERENCES

[1] K. P. BELKHALE AND P. BANERJEE, *Approximate algorithms for the partitionable independent task scheduling problem*, Proc. International Conference on Parallel Processing, August 1990, Vol. I, pp. 72–75.

[2] J. BLAZWICZ, M. DRABOWSKI, AND J. WEGLARZ, *Scheduling multiprocessor tasks to minimize schedule length*, IEEE Trans. Comput., 35 (1986), pp. 389–393.

[3] G.-I. CHEN AND T.-H. LAI, *Scheduling independent jobs on hypercubes*, Proc. Conference Theoretical Aspects of Computer Science, 1988, pp. 273–280.

[4] Y. CHO AND S. SAHNI, *Bounds for list schedules on uniform processors*, SIAM J. Comput., 9 (1980), pp. 91–103.

[5] J. DU AND J. Y.-T. LEUNG, *Complexity of scheduling parallel task systems*, SIAM J. Discrete Math., 2 (1989), pp. 473–487.

[6] T. GONZALES, O. H. IBARRA, AND S. SAHNI, *Bounds for* LPT *schedules on uniform processors*, SIAM J. Comput., 6 (1977), pp. 155–166.

[7] R.L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[8] R. L. GRAHAM, E. L. LAWLER, J. K. LENSTRA, AND A. H. G. RINNOOY KAN, *Optimization and approximation in deterministic sequencing and scheduling: A survey*, Ann. Discrete Math., 15 (1979), pp. 287–326.

[9] R. KRISHNAMURTI AND E. MA, *The processor partitioning problem in special-purpose partitionable systems*, Proc. International Conference on Parallel Processing, August 1988, Vol. I, pp. 434–443.

[10] B. VELTMAN, B. J. LAGEWEG, AND J. K. LENSTRA, *Multiprocessor scheduling with communication delays*, Report BS-R 9018, Mathematical Centre, Amsterdam, June 1990.

[11] Q. WANG AND K. H. CHENG, *List scheduling of parallel tasks*, Inform. Process. Lett., 37 (1991), pp. 291–297.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 295–315, April 1992
©1992 Society for Industrial and Applied Mathematics
007

HEURISTIC SAMPLING: A METHOD FOR PREDICTING THE PERFORMANCE OF TREE SEARCHING PROGRAMS*

PANG C. CHEN†

Abstract. Determining the feasibility of a particular search program is important in practical situations, especially when the computation involved can easily require days, or even years. To help make such predictions, a simple procedure based on a stratified sampling approach is presented. This new method, which is called heuristic sampling, is a generalization of Knuth's original algorithm for estimating the efficiency of backtrack programs. With the aid of simple heuristics, this method can produce significantly more accurate cost estimates for commonly used tree search algorithms such as depth-first, breadth-first, best-first, and iterative-deepening.

Key words. heuristics, stratified sampling, search tree, feasibility testing, cost estimation, Monte Carlo method, analysis of algorithms

AMS(MOS) subject classifications. 68Q25, 65C05

1. Introduction. Tree searching [8] is a general, easily implemented problem-solving technique. Unfortunately, the efficiency of tree searching programs is usually difficult to analyze, even at a rudimentary level. Without analytic cost information, the typical course is to let the computer run until it either finishes the job or exhausts our patience. Switching to a more computational sampling approach provides a less haphazard alternative: We gain accuracy in understanding particular search programs and thus design better ones.

The sampling method that we will discuss generalizes an algorithm of Knuth [5] for estimating properties of a backtrack tree. Starting at the root, Knuth's algorithm extends a partial path by expanding the end node and picking a child according to a uniform distribution. It then forms an unbiased estimate of the tree property using the branching degrees along the randomly selected path. According to Knuth, this simple estimation procedure worked consistently well in his experiments. But as a refinement, Knuth also suggested the technique of importance sampling [2] in which the child is selected according to a weighted distribution, with the weight of each child being an estimate of the property of the corresponding subtree.

Unsatisfied with the results of Knuth's algorithm in his own experiments, Purdom [9] later modified the algorithm in an attempt to reduce variance. The modified algorithm allows more than one child to be considered for further exploration, and it is therefore called partial backtracking. If only one child per parent is explored, then Purdom's algorithm reduces to Knuth's algorithm; but if every child is explored, then the estimate becomes exact due to a complete backtrack search. The algorithm offers a varying degree of efficiency that depends on the extent of exploration. However, it is difficult to reduce variance significantly without incurring high experimental cost.

Heuristic sampling generalizes Knuth's algorithm in a different direction. Instead of taking the importance sampling approach, the method adopts another statistical technique known as stratified sampling [2], based on a "heuristic function" to be supplied by the algorithm designer. This heuristic function, which we call a stratifier, should in principle reflect the broad characteristics of the nodes in the search tree. If two nodes have similar features, then they should be classified into the same stratum. By exploiting the tree structure as reflected through the stratifier, stratified sampling on the search tree

*Received by the editors April 16, 1990; accepted for publication May 4, 1991. The work was performed at Stanford University, where it was supported by a Hertz Fellowship, and at Sandia National Laboratories, where it was supported by U.S. Department of Energy contract DE-AC04-76DP00789.

†Sandia National Laboratories, Albuquerque, New Mexico 87185.

reduces variance relative to Knuth's algorithm, and limits the cost of prediction to the number of strata.

To present heuristic sampling, we first introduce the concept of a stratifier, and then discuss a sampling scheme that employs stratifiers to estimate tree properties. We examine the accuracy and cost of the sampling scheme both analytically and empirically. In particular, we demonstrate the effectiveness of heuristic sampling analytically with a random graph problem, and empirically with a chessboard recreation.

2. Problem formulation. Let $\boldsymbol{T}$ be a tree with nodes (or states) $\boldsymbol{S}$, and let f be an arbitrary function on $\boldsymbol{S}$. Then, abstractly, the goal of heuristic sampling is to produce estimates of the sum

$$\varphi \stackrel{\text{def}}{=} \sum_{s \in \boldsymbol{S}} f(s) \tag{1}$$

without traversing the entire tree. The value of φ is a property of $\boldsymbol{T}$, as captured by the definition of f. For instance, if $f(s)$ is the cost of processing node s, then φ becomes the total cost of traversing $\boldsymbol{T}$; if $f(s)$ is the number of children of s, then φ becomes the total number of edges in $\boldsymbol{T}$; and if f is an indicator function for some property $\mathcal{X}$, then φ becomes the number of nodes satisfying $\mathcal{X}$.

The performance of some alternative heuristic search algorithms can be studied by estimating φ, using an appropriate choice of f. For example, suppose that F is a monotone heuristic cost function [8], and that $\mathcal{X}$ represents the property $F(s) \leq F_0$. Then estimating $\varphi(F_0)$ predicts the number of nodes that a best-first search will have to examine before it can process any other node with cost $> F_0$. Likewise, estimating the sum $\sum_{F_1 \leq F_0} \varphi(F_1)$ predicts the respective complexity for iterative deepening [6].

3. Heuristic sampling. Our sampling technique requires a stratifier: a heuristic function $h : \boldsymbol{S} \rightarrow \mathcal{P}$ that maps each state s to a stratum $h(s)$ in a partially ordered set $\mathcal{P}$, with h strictly decreasing along each edge of $\boldsymbol{T}$. (Notice that we can always construct a stratifier by concatenating the depth of the node along with any other state information into a vector representing the stratum, and using a lexicographic ordering to compare these strata.) Given that $\boldsymbol{T}$ is stratified under h, our idea is to sample each stratum α for a representative node s_α and simultaneously to obtain an estimate $w_\alpha = w(s_\alpha)$ for the number of nodes in that stratum. From the sample set of nodes, an unbiased estimate of φ can then be obtained by computing

$$\widehat{\varphi} \stackrel{\text{def}}{=} \sum_{\alpha} w_\alpha f(s_\alpha). \tag{2}$$

Figure 1 gives an algorithm (HS) for finding the sample nodes and weights needed to compute $\widehat{\varphi}$. The algorithm explores at most one node for each stratum and pretends that all other nodes in the stratum have the same subtree structure. It initializes the weight of the root to 1, and inserts the node-weight pair into a queue for further exploration. The queue is maintained with at most one representative from each stratum. The procedure is to remove and expand the node in the front of the queue (the one in a maximal stratum) until the queue becomes empty. To prevent multiple representatives, each child is given a chance to replace the current representative in the queue. The random replacement is implemented in such a way that a child will be expanded with probability proportional to the weight of its parent.

Both the accuracy and the time complexity of the sampling scheme depend on the stratifier. The variance of the estimator depends on the homogeneity of each stratum,

```
Algorithm HS
input: root of a tree and a stratifier h;
output: set of sample nodes S and corresponding weights W;
data structure: a queue Q with elements (s, w) indexed by h(s);
begin
  Q ← {(root, 1)};
  while (Q not empty) do
        output an element (s, w) of Q with maximal h(s);
        for each child t of s do
            α ← h(t);
            if Q contains an element (s_α, w_α) in stratum α then do
              w_α ← w_α + w;
              with probability w/w_α do
                s_α ← t;
            else
              insert a new element (t, w) into Q;
end.
```

FIG. 1. *Heuristic sampling.*

which we will discuss in the analysis section. As for the sampling time, notice that we can guarantee an output of at most one representative node per stratum because we choose to process from the higher stratum down. As a result, the number of nodes that need to be expanded by the sampling scheme is bounded by the number of strata.

3.1. An example. Consider the simple problem of determining the number of paths from (m, n) to $(0, 0)$ on the Cartesian plane, where the only legal moves are to the left or downward in steps of unit length. This problem is well known and the exact answer is $\binom{m+n}{m}$. Nevertheless, suppose that we do not know how to solve the problem analytically. Then a naïve approach is to enumerate a tree with each node corresponding to a partial legal path starting (m, n), and each edge corresponding to a legal move. In this tree, the number of leaves is the desired answer. To estimate, apply heuristic sampling with f being 1 on each leaf, and zero everywhere else. If we stratify according to

$$h_0(s) = -\text{depth}(s), \tag{3}$$

then the sampling procedure becomes Knuth's algorithm. Observe that in the absence of additional knowledge, we can always try this depth stratification. In fact, we can also try

$$g_0(s) = \langle -\text{depth}(s), \text{children}(s) \rangle, \tag{4}$$

which can perhaps give us a better stratification. Still, for our problem, there is a much better stratifier that exploits additional knowledge; we can stratify according to

$$h(s) = \langle i, j \rangle \tag{5}$$

with $\langle i, j \rangle$ being the end point of the partial path corresponding to s. Under this stratification, the estimator $\widehat{\varphi}$ is exact (unbiased with zero variance), because for each stratum the number of leaves in each subtree is the same. Therefore, the tree size $\binom{m+n}{m}$ is estimated exactly by expanding only $(m+1)(n+1)$ nodes, even though $\binom{m+n}{m}$ is exponentially large.

3.2. Adding importance sampling. We will show that $\widehat{\varphi}$ remains unbiased when we alter the child selection probability by replacing the clause

$$w_\alpha \leftarrow w_\alpha + w;$$
$$\textbf{with probability } w/w_\alpha \textbf{ do}$$
$$s_\alpha \leftarrow t;$$

in heuristic sampling (HS) with

$$w_\alpha \leftarrow w_\alpha/(1-p);$$
$$\textbf{with probability } p \textbf{ do}$$
$$s_\alpha \leftarrow t;$$
$$w_\alpha \leftarrow w/p;$$

using an arbitrary importance probability $0 < p < 1$ in each iteration. We call the resulting generic sampling algorithm GS. As with Knuth's algorithm [5], there exists a sequence of p's that will force $\widehat{\varphi}$ to be exactly equal to φ, provided that f is nonnegative. In practice, however, this sequence may be difficult to find.

4. Analysis. We will now examine the behavior of HS, GS (which incorporates importance sampling), and simple sampling (SS) of Knuth [5] that uses depth as the stratifier. Because GS is a generalization of HS, which is in turn a generalization of SS, anything we prove for GS also holds for HS, and anything that is true for HS is also true for SS.

4.1. Preliminaries. Let $\alpha_0 = h(\text{root})$ be the maximum stratum and denote the parent of s by $\hat{s}$. From the definition of a stratifier, we know that the set of expanded nodes S in the output can have at most one node per stratum; hence we can write

$$S = \bigcup_{\alpha \in \mathcal{P}} \{s_\alpha\} \tag{6}$$

with s_α being the unique representative node for stratum α, if it exists. Let W be the set of weights produced along with S, and let $w(s_\alpha) = w_\alpha$. To prevent any ambiguity in selecting a maximal element of Q, we assume that the strata are processed according to a total ordering $\succ$, consistent with $\mathcal{P}$. Later we show that the output of HS is statistically invariant under any particular total ordering of $\mathcal{P}$. Thus, define

$$S_\alpha \stackrel{\text{def}}{=} \bigcup_{\beta \succ \alpha} \{s_\beta\} \tag{7}$$

to be the intermediate output before processing stratum α, and denote the substratum of α (the one immediately below) by $\underline{\alpha}$.

4.2. Sample trees. According to GS, every intermediate output S_α must form a subtree T_α because for any nonroot node to be inserted into Q, its parent must have already been expanded. In particular, the final output S must also form a subtree T, which we call a sample tree.

From the sampling procedure, we know that a sample tree T must contain the parent of every nonroot node of T and a stratum representative for every child of every nonleaf

node of T. Conversely, any subtree T of $\boldsymbol{T}$ satisfying the condition above must also have a positive probability of being generated by GS. The output probability $\mathbf{P}(T)$ associated with each instance of T is related directly to the weights of the nodes produced.

THEOREM 1. *The probability that* GS *generates sample tree T with weights W is*

$$\mathbf{P}(T) = \prod_{(s,t)\in T} \frac{w(s)}{w(t)}. \tag{8}$$

Under HS, *these weights satisfy the recurrence*

$$w_{\alpha_0} = 1; \qquad w_\alpha = \sum_{\beta \succ \alpha} w_\beta c_\alpha(s_\beta) \quad \text{for } \alpha \prec \alpha_0, \tag{9}$$

where $c_\alpha(s)$ denotes the number of children of s in stratum α.

Proof. Given T_α, the probability of leaving stratum α with $s_\alpha = t$ is equal to $p\prod_i q_i$, where p is the probability of choosing t to replace the current queue element, and q_i is the probability of not making the ith replacement of t with t being in the queue. These probabilities are just ratios of weights by construction, and their product telescopes to $w(\hat{t})/w(t)$. On the other hand, if s_α does not exist, then $T_{\underline{\alpha}} = T_\alpha$ and $\mathbf{P}(s_\alpha \mid T) = 1$ vacuously. Hence, unfolding the recurrence

$$\mathbf{P}(T_{\underline{\alpha}}) = \mathbf{P}(T_\alpha)\mathbf{P}(s_\alpha \mid T_\alpha) \tag{10}$$

yields the desired product for $\mathbf{P}(T)$.

Under HS, the weight w_α for $\alpha \prec \alpha_0$ is simply equal to the sum of $w(\hat{t})$ for all t with $h(t) = \alpha$ and $\hat{t}$ in T, a sum which is clearly equivalent to the desired formula. □

COROLLARY 2. *The output probability of a sample tree of* HS *is independent of the total ordering imposed on $\mathcal{P}$. Consequently, which maximal element we choose to expand at each stage has no effect on the distribution of $\widehat{\varphi}$.*

Proof. The recurrence governing the weights is invariant under different total orderings of $\mathcal{P}$. □

For the SS case, every sample tree is just a (sample) path $\rho(s)$ from the root to a leaf s.

COROLLARY 3. *Under* SS, *the probability of producing a sample path $T = \rho(s)$ with length k is*

$$\mathbf{P}(\rho(s)) = 1/w(s) = 1/\prod_{0<j\leq k} c(s^j), \tag{11}$$

where $c(s)$ denotes the number of children of s, and s^j denotes the jth ancestor of s, with $s^1 = \hat{s}$.

4.3. Unbiasedness of $\widehat{\varphi}$. The unbiasedness of $\widehat{\varphi}$ as produced from GS can be proved quite easily with concepts from martingale theory [4]. During the expansion of s_α, define $Q_{\alpha,k}$ to be the set of representatives in the queue before considering the kth child of s_α, and define $C_{\alpha,k}$ to be those children of s_α yet to be considered. Also define $\boldsymbol{T}(s)$ to be the induced subtree of s in $\boldsymbol{T}$, and define

$$\Phi(s) \stackrel{\text{def}}{=} \sum_{t\in \boldsymbol{T}(s)} f(s). \tag{12}$$

THEOREM 4. *The estimator $\widehat{\varphi}$ under* GS *is unbiased in that*

$$\mathbf{E}\widehat{\varphi} = \Phi(\text{root}) = \varphi. \tag{13}$$

Proof. The appropriate statistic to study at each iteration of GS is the successive approximation

$$\varphi^{(\alpha,k)} \stackrel{\text{def}}{=} \sum_{u \in Q_{\alpha,k}} w(u)\Phi(u) + \sum_{t \in C_{\alpha,k}} w_\alpha \Phi(t) + \sum_{\beta \succeq \alpha} w_\beta f(s_\beta), \tag{14}$$

which is completely determined when given the data

$$Z_{\alpha,k} \stackrel{\text{def}}{=} S_{\underline{\alpha}} \cup Q_{\alpha,k} \cup C_{\alpha,k}. \tag{15}$$

If $k \leq c(s_\alpha)$ and the kth child t of s_α is supposed to replace the current node u in the queue with probability $p = 1-q$, then the expected value of the successive approximation remains the same after the random replacement, for we have

$$\mathbf{E}(\varphi^{(\alpha,k+1)} - \varphi^{(\alpha,k)} \mid Z_{\alpha,k}) = p\frac{w_\alpha}{p}\Phi(t) + q\frac{w(u)}{q}\Phi(u) - w_\alpha\Phi(t) - w(u)\Phi(u) = 0. \tag{16}$$

On the other hand, if $k > c(s_\alpha)$ and node u in the queue is supposed to be removed and expanded next, then the successive approximation also remains the same after the expansion of u, for we have

$$\mathbf{E}(\varphi^{(\underline{\alpha},1)} - \varphi^{(\alpha,k)} \mid Z_{\alpha,k}) = w(u)f(u) + w(u) \sum_{(u,t) \in \boldsymbol{T}} \Phi(t) - w(u)\Phi(u) = 0. \tag{17}$$

Consequently, the expected value of our successive approximation at the end of the sampling ($\widehat{\varphi}$) is equal to its initial value ($\Phi(\text{root}) = \varphi$). □

4.4. Conditions for minimum variance. If f is nonnegative, then in the preceding proof we can choose to replace the current node u with the kth child t of s with probability

$$p = w(s)\Phi(t)/(w(s)\Phi(t) + w(u)\Phi(u)) \tag{18}$$

to ensure $\varphi^{(\alpha,k+1)} = \varphi^{(\alpha,k)}$. The resulting sequence of p's is exactly the one we need to reduce variance of $\widehat{\varphi}$ to zero. Of course, computing this perfect sequence is not feasible in practice. But if we model our heuristic stratifier by assuming that the subtrees corresponding to each stratum are drawn from the same distribution of trees, then we can show that the probability $p = w(s)/(w(s) + w(u))$ used in HS is indeed the one that minimizes the average variance of $\widehat{\varphi}$.

THEOREM 5. *Suppose that f is nonnegative, and that we are running* GS *on a random tree $\boldsymbol{T}$ in which the subtrees corresponding to each stratum are drawn from the same distribution. Then, $\mathbf{EV}(\widehat{\varphi} \mid \boldsymbol{T})$, the average variance of $\widehat{\varphi}$, is minimized if at every random replacement stage, we choose to replace the current node u with a child of s with probability $p = w(s)/(w(s) + w(u))$.*

Proof. Consider the effect of varying p for one particular random replacement stage and leaving everything else fixed. We have

$$\widehat{\varphi} = \begin{cases} Aw(s)/p + B & \text{with probability } p, \\ Aw(u)/(1-p) + B & \text{with probability } 1-p, \end{cases} \tag{19}$$

where $A \geq 0$ and B are independent of p. Hence, the variance of $\widehat{\varphi}$ can be written as

$$A'(w^2(s)/p + w^2(u)/(1-p)) + B', \tag{20}$$

with $A' \geq 0$ and B' independent of p. Taking the average over all random trees and differentiating with respect to p yields the formula

$$-A''(w^2(s)/p^2 - w^2(u)/(1-p)^2) \tag{21}$$

with $A'' \geq 0$. Setting this derivative to zero, we find that p has to be proportional to $w(s)$, implying that $p = w(s)/(w(s)+w(u))$. The nonnegativity of the second derivative shows that this condition does indeed minimize the average variance. □

4.5. Variance of $\widehat{\varphi}$. To derive a formula for $\mathbf{V}\widehat{\varphi}$, we use the superscript (α) on an operator to denote the conditional operator given S_α. We also use ∂S_α to denote the children of S_α in stratum $\preceq \alpha$, and use $R_\alpha = R(S_\alpha)$ to denote the nodes of ∂S_α in stratum α. Notice that R_α contains precisely the candidates for s_α, and depending on which $t \in R_\alpha$ is selected as s_α, the weight w_α is equal to $w(\hat{t})$ divided by $\mathbf{P}^{(\alpha)}(t)$, the probability of choosing t.

For $\alpha \prec \alpha_0$, consider the successive approximation

$$\varphi^{(\alpha)} \stackrel{\text{def}}{=} \sum_{\beta \succ \alpha} w_\beta f(s_\beta) + \sum_{s \in \partial S_\alpha} w(\hat{s})\Phi(s), \tag{22}$$

which depends only on S_α. At the initial stage when α corresponds to $\underline{\alpha_0}$, we have $S_\alpha = \{\text{root}\}$ and $\varphi^{(\alpha)} = \varphi$; at the final stage when α corresponds to the lowest stratum, we have $S_{\underline{\alpha}} = S$ and $\varphi^{(\alpha)} = \widehat{\varphi}$. Because of this link, we can characterize the behavior of $\widehat{\varphi}$ by studying the changes in $\varphi^{(\alpha)}$ as α moves through $\mathcal{P}$.

LEMMA 6.

$$\mathbf{E}^{(\alpha)}\varphi^{(\underline{\alpha})} = \varphi^{(\alpha)}. \tag{23}$$

Proof. The set $\partial S_{\underline{\alpha}}$ is obtained from ∂S_α by removing R_α and adding the children of s_α; hence

$$\varphi^{(\underline{\alpha})} - \varphi^{(\alpha)} = w_\alpha f(s_\alpha) - \sum_{t \in R_\alpha} w(\hat{t})\Phi(t) + \sum_{(s_\alpha, v) \in \boldsymbol{T}} w(s_\alpha)\Phi(v) \tag{24}$$

$$= w_\alpha \Phi(s_\alpha) - \sum_{t \in R_\alpha} w(\hat{t})\Phi(t). \tag{25}$$

But we also know that

$$\mathbf{E} w_\alpha \Phi(s_\alpha) = \sum_{t \in R_\alpha} \mathbf{P}^{(\alpha)}(t) \frac{w(\hat{t})}{\mathbf{P}^{(\alpha)}(t)} \Phi(t). \tag{26}$$

Therefore, we have $\mathbf{E}^{(\alpha)}(\varphi^{(\underline{\alpha})} - \varphi^{(\alpha)}) = 0$. □

LEMMA 7.

$$\mathbf{V}\varphi^{(\underline{\alpha})} = \mathbf{V}\varphi^{(\alpha)} + \mathbf{E}\mathbf{V}^{(\alpha)} w_\alpha \Phi(s_\alpha). \tag{27}$$

Proof. Conditioning on S_α, we have

$$\mathbf{V}\varphi^{(\underline{\alpha})} = \mathbf{V}\mathbf{E}^{(\alpha)}\varphi^{(\underline{\alpha})} + \mathbf{E}\mathbf{V}^{(\alpha)}\varphi^{(\underline{\alpha})} \tag{28}$$

$$= \mathbf{V}\varphi^{(\alpha)} + \mathbf{E}\mathbf{V}^{(\alpha)}\varphi^{(\underline{\alpha})} \tag{29}$$

according to the previous lemma. The desired result then follows since

$$\mathbf{V}^{(\alpha)}\varphi^{(\underline{\alpha})} = \mathbf{V}^{(\alpha)}(\varphi^{(\underline{\alpha})} - \varphi^{(\alpha)}) \tag{30}$$

$$= \mathbf{V}^{(\alpha)} w_\alpha \Phi(s_\alpha). \tag{31}$$

□

THEOREM 8. *Under* GS,

$$\mathbf{V}\widehat{\varphi} = \sum_{\alpha \prec \alpha_0} \mathbf{E}\mathbf{V}^{(\alpha)} w_\alpha \Phi(s_\alpha). \tag{32}$$

Proof. Simply unfold the recurrence of the lemma above, and identify $\varphi^{(\underline{a})}$ with $\widehat{\varphi}$, assuming that a is the lowest stratum in $\mathcal{P}$. □

For HS, the weight $w_\alpha = \sum_{t \in R_\alpha} w(\hat{t})$ is independent of the choice made on s_α, so we can move w_α out of $\mathbf{V}^{(\alpha)}$ and obtain the following theorem.

THEOREM 9. *Under* HS,

$$\mathbf{V}\widehat{\varphi} = \sum_{\alpha \prec \alpha_0} \mathbf{E} w_\alpha^2 \mathbf{V}^{(\alpha)} \Phi(s_\alpha). \tag{33}$$

For SS, the weight $w(s)$ is simply the product of branching degrees of the ancestors of s.

THEOREM 10. *Under* SS,

$$\mathbf{V}\widehat{\varphi} = \sum_{s \in \boldsymbol{S}} c^2(s) w(s) V(s), \tag{34}$$

where $V(s)$ *denotes the mean-square deviation of the* Φ *value of its children from the average* Φ *value of its children.*

Proof. If $T_\alpha = \rho(s)$, then $\mathbf{P}(T_\alpha) = 1/w(s)$, and $w_\alpha = w(s)c(s)$, implying that

$$\mathbf{V}\widehat{\varphi} = \textstyle\sum_{s \in \boldsymbol{S}} \frac{1}{w(s)} w^2(s) c^2(s) V(s). \tag{35}$$

□

4.6. Homogeneity of strata. The intuitive qualification of a good stratifier h is that it should provide some degree of homogeneity among nodes in the same stratum. We formalize this concept of homogeneity by introducing a measure on its counterpart. For each stratum α, define the parameter

$$\widetilde{\nu}_\alpha \stackrel{\text{def}}{=} \max_{S_\alpha} \frac{\mathbf{V}^{(\alpha)}\Phi(s_\alpha)}{(\mathbf{E}^{(\alpha)}\Phi(s_\alpha))^2}, \tag{36}$$

and call the parameters collectively as the degree of variation $\widetilde{\nu}$ with respect to Φ under h. Notice that the sequence $\widetilde{\nu}$ is always nonnegative, with the zero sequence indicating total homogeneity within each stratum.

The following theorem demonstrates a relationship between the degree of homogeneity and the accuracy of $\widehat{\varphi}$.

THEOREM 11. *Suppose that f is nonnegative and that the tree $\boldsymbol{T}$ under stratifier h has degree of variation $\widetilde{\nu}$ with respect to Φ. Then*

$$\mathbf{V}\widehat{\varphi} \leq \left(\prod_{\alpha \prec \alpha_0} (1 + \widetilde{\nu}_\alpha) - 1 \right) \varphi^2 \tag{37}$$

under HS.

Proof. From the proof of Lemma 7, we know that

$$\mathbf{V}^{(\alpha)} \varphi^{(\underline{\alpha})} = w_\alpha^2 \mathbf{V}^{(\alpha)} \Phi(s_\alpha), \tag{38}$$

which is by definition less than or equal to $\widetilde{\nu}_\alpha w_\alpha^2 (\mathbf{E}^{(\alpha)} \Phi(s_\alpha))^2$. On the other hand, we also know that

$$\mathbf{E}^{(\alpha)} w_\alpha \Phi(s_\alpha) = \sum_{t \in R_\alpha} w(\hat{t}) \Phi(t) \leq \varphi^{(\alpha)}, \tag{39}$$

because $R_\alpha \subseteq \partial S_\alpha$. Hence, we have

$$\begin{aligned} \mathbf{E}^{(\alpha)} (\varphi^{(\underline{\alpha})})^2 &= (\mathbf{E}^{(\alpha)} \varphi^{(\underline{\alpha})})^2 + \mathbf{V}^{(\alpha)} \varphi^{(\underline{\alpha})} \\ &\leq (\mathbf{E}^{(\alpha)} \varphi^{(\underline{\alpha})})^2 + \widetilde{\nu}_\alpha (\varphi^{(\alpha)})^2 \\ &= (1 + \widetilde{\nu}_\alpha)(\varphi^{(\alpha)})^2, \end{aligned}$$

since $\mathbf{E}^{(\alpha)} \varphi^{(\underline{\alpha})} = \varphi^{(\alpha)}$. Consequently, we have the inequality

$$\mathbf{E}(\varphi^{(\underline{\alpha})})^2 \leq (1 + \widetilde{\nu}_\alpha) \mathbf{E}(\varphi^{(\alpha)})^2. \tag{40}$$

Unfolding this inequality and identifying $\varphi^{(\underline{\alpha}_0)}$ with φ then yields the theorem. □

An immediate consequence of this theorem is that if the parameters $\widetilde{\nu}_\alpha$ are all fairly small, say, less than some quantity ϵ, with $\epsilon \, \|h(\boldsymbol{S})\| = o(1)$, then the normalized variance $\mathbf{V}\widehat{\varphi}/\varphi^2$ can be at most

$$\epsilon \, \|h(\boldsymbol{S})\| + O(\epsilon \, \|h(\boldsymbol{S})\|)^2. \tag{41}$$

However, if ϵ is not small enough, then the bound can grow exponentially in the number of strata. Fortunately, with more careful bookkeeping, we can obtain a sharper bound in which the growth is only exponential in the depth of the tree, rather than in the total number of strata.

To derive this bound, we introduce the concept of layered poset partitioning. A poset partition $\mathcal{P} = \bigcup_i A_i$ is said to be layered if the A_i's are disjoint antichains with $A_i \prec A_j$ for all $j < i$. In other words, every pair of strata in A_i should be incomparable (in the sense that no member of one stratum is a descendent of another in the other stratum) and no stratum in A_i should be higher than any stratum in A_j for $j < i$. Think of A_j as the set of strata with depth j. With respect to our previous enforcement of a total ordering on $\mathcal{P}$, each A_i is simply a stratum, and each stratum are processed in the corresponding order. Under the more general layered partitioning of $\mathcal{P}$, each antichain is also processed in order. But because every pair of strata in an antichain is incomparable, the corresponding strata are processed independently. Exploiting this property yields the following generalization.

THEOREM 12. *Suppose that f is nonnegative and that the state space $\boldsymbol{S}$ under the stratifier has degree of variation $\widetilde{\nu}$ with respect to Φ. Then the variance of $\widehat{\varphi}$ under* HS *is bounded above by*

$$\left(\prod_{0<i\leq r}\left(1+\max_{\alpha\in A_i}\widetilde{\nu}_\alpha\right)-1\right)\varphi^2, \tag{42}$$

where $\bigcup_{0<i<r} A_i$ is an arbitrary layered partition of $\mathcal{P}$.

Proof. We take the same approach as in the previous theorem. However, instead of working with random variables conditioning on S_α, we now condition them on

$$S_i \overset{\text{def}}{=} \bigcup_{j<i}\bigcup_{\alpha\in A_j}\{s_\alpha\}, \tag{43}$$

and use the superscript (i) on an operator to denote the conditional operator given S_i. Thus, consider the conditional random variable

$$\varphi_i \overset{\text{def}}{=} \mathbf{E}^{(i)}\widehat{\varphi} = \sum_{s\in S_i} w(s)f(s) + \sum_{s\in\partial S_i} w(\hat{s})\Phi(s), \tag{44}$$

where ∂S_i, analogous to ∂S_α, is defined to be the children of S_i in strata lower than those in S_i. Notice that as we extend the conditional information from S_i to S_{i+1}, the only part of $\widehat{\varphi}$ that varies is the sum $\sum_{\alpha\in A_i} w_\alpha\Phi(s_\alpha)$. This result follows because for each α in A_i, the parents of the prospective nodes for s_α are all in S_i. Incidentally, this fact tells us that the weight $w(s_\alpha)$ depends solely on S_i, and not on $S_{i+1}\setminus S_i$. Hence, we have

$$\mathbf{V}^{(i)}\varphi_{i+1} = \mathbf{V}^{(i)}\sum_{\alpha\in A_i} w_\alpha\Phi(s_\alpha) = \sum_{\alpha\in A_i} w_\alpha^2\mathbf{V}^{(i)}\Phi(s_\alpha), \tag{45}$$

since each s_α is selected independently. Applying the definition of $\widetilde{\nu}$, we obtain the upper bound

$$\sum_{\alpha\in A_i}\widetilde{\nu}_\alpha w_\alpha^2(\mathbf{E}^{(i)}\Phi(s_\alpha))^2. \tag{46}$$

But

$$w_\alpha^2(\mathbf{E}^{(i)}\Phi(s_\alpha))^2 = \left(\sum_{t\in R_\alpha(S_i)} w(\hat{t})\Phi(t)\right)^2, \tag{47}$$

with $R_\alpha(S_i)$ being those children in ∂S_i in stratum α. Again, because the strata in A_i are incomparable, the sets $R_\alpha(S_i)$ are disjoint between different α's. This conclusion yields the bound

$$\sum_{\alpha\in A_i} w_\alpha^2(\mathbf{E}^{(i)}\Phi(s_\alpha))^2 \leq \left(\sum_{t\in R_i} w(\hat{t})\Phi(t)\right)^2, \tag{48}$$

where R_i is the union of $R_\alpha(S_i)$ over α. Finally, we also have

$$\varphi_i \geq \sum_{t\in R_i} w(\hat{t})\Phi(t), \tag{49}$$

because $R_i \subseteq \partial S_i$. Combining all these facts, we get

$$\mathbf{V}^{(i)}\varphi_{i+1} \le \max_{\alpha\in A_i} \widetilde{\nu}_\alpha \varphi_i^2, \tag{50}$$

which implies that

$$\mathbf{E}^{(i)}\varphi_{i+1}^2 \le \left(1 + \max_{\alpha\in A_i} \widetilde{\nu}_\alpha\right)\varphi_i^2, \tag{51}$$

and consequently that

$$\mathbf{E}\varphi_{i+1}^2 \le (1 + \max_{\alpha\in A_i} \widetilde{\nu}_\alpha)\mathbf{E}\varphi_i^2. \tag{52}$$

The desired bound is now obtained by unfolding this inequality and identifying φ_1 with φ. □

The preceding technique can also be applied to analyze the average variance of $\widehat{\varphi}$ when the input tree $\boldsymbol{T}$ is random. Let the superscripts (i) and $\{i\}$ on an operator denote, respectively, the conditional operator given S_i, and the conditional operator given S_i and $\boldsymbol{T}$. Suppose that $\Phi(s)$ for each s in stratum α shares (but not necessarily independently) the same distribution Φ_α. Then instead of $\widetilde{\nu}$, we can express a bound on $\mathbf{EV}(\widehat{\varphi} \mid \boldsymbol{T})$ in terms of the average variation $\bar{\nu}$, whose component corresponding stratum α is defined as

$$\bar{\nu}_\alpha \stackrel{\text{def}}{=} \mathbf{V}\Phi_\alpha/\mathbf{E}^2\Phi_\alpha. \tag{53}$$

LEMMA 13.

$$\mathbf{E}^{(i)}\mathbf{V}^{\{i\}}\Phi(s_\alpha) \le \mathbf{V}\Phi_\alpha. \tag{54}$$

Proof. The conditional variable $\mathbf{V}^{\{i\}}\Phi(s_\alpha)$ can be put in the form

$$\sum_j p_j\Phi(s_j)^2 - \left(\sum_j p_j\Phi(s_j)\right)^2, \tag{55}$$

where p_j is the probability that we choose s_j to be s_α. Taking the conditional expectation $\mathbf{E}^{(i)}$ of the first sum, we get $\sum_j p_j\mathbf{E}\Phi_\alpha^2 = \mathbf{E}\Phi_\alpha^2$. Let X^2 be the second term. By the nonnegativity of the variance, we have $\mathbf{E}^{(i)}X^2 \ge (\mathbf{E}^{(i)}X)^2$. But $\mathbf{E}^{(i)}X = \sum_j p_j\mathbf{E}\Phi_\alpha = \mathbf{E}\Phi_\alpha$. Combining all of the facts, we have

$$\mathbf{E}^{(i)}\mathbf{V}^{\{i\}}\Phi(s_\alpha) \le \mathbf{E}\Phi_\alpha^2 - (\mathbf{E}\Phi_\alpha)^2, \tag{56}$$

which implies the lemma. □

THEOREM 14. *Suppose that f is nonnegative and that the random tree $\boldsymbol{T}$ under the stratifier has average degree of variation $\bar{\nu}$ with respect to Φ. Then, under* HS, *the average variance of $\widehat{\varphi}$ over the random input trees is bounded above by*

$$\left(\prod_{0<i\le r}\left(1 + \max_{\alpha\in A_i}\bar{\nu}_\alpha\right) - 1\right)\mathbf{E}^2\varphi, \tag{57}$$

where $\bigcup_{0\leq i\leq r} A_i$ *is an arbitrary layered partition of* $\mathcal{P}$.

Proof. Again, consider the conditional variable

$$\varphi_i \stackrel{\text{def}}{=} \mathbf{E}^{\{i\}}\widehat{\varphi} = \sum_{s\in S_i} w(s)f(s) + \sum_{s\in\partial S_i} w(\hat{s})\Phi(s). \tag{58}$$

We have

$$\mathbf{E}^{(i)}\mathbf{V}^{\{i\}}\varphi_{i+1} = \sum_{\alpha\in A_i} w_\alpha^2 \mathbf{E}^{(i)}\mathbf{V}^{\{i\}}\Phi(s_\alpha) \tag{59}$$

$$\leq \sum_{\alpha\in A_i} w_\alpha^2 \mathbf{V}\Phi_\alpha^2 \tag{60}$$

$$\leq \sum_{\alpha\in A_i} \bar{\nu}_\alpha (\mathbf{E}^{(i)} w_\alpha \Phi_\alpha)^2 \tag{61}$$

$$\leq \left(\max_{\alpha\in A_i} \bar{\nu}_\alpha\right) \sum_{\alpha\in A_i} (\mathbf{E}^{(i)} w_\alpha \Phi_\alpha)^2, \tag{62}$$

where the second step follows from the preceding lemma, and the third step follows from the definition of $\bar{\nu}$. To relate this inner sum with $\mathbf{E}^{(i)}\varphi_i$, notice that

$$\sum_{\alpha\in A_i} (\mathbf{E}^{(i)} w_\alpha \Phi_\alpha)^2 \leq \sum_{\alpha} \left(\sum_{t\in R_\alpha(S_i)} w(\hat{t})\mathbf{E}^{(i)}\Phi(t)\right)^2 \tag{63}$$

$$\leq \left(\sum_{t\in R_i} w(\hat{t})\mathbf{E}^{(i)}\Phi(t)\right)^2 \tag{64}$$

$$\leq (\mathbf{E}^{(i)}\varphi_i)^2 \tag{65}$$

$$\leq \mathbf{E}^{(i)}\varphi_i^2. \tag{66}$$

Consequently, we have

$$\mathbf{E}\varphi_{i+1}^2 \leq \left(1 + \max_{\alpha\in A_i} \bar{\nu}_\alpha\right) \mathbf{E}\varphi_i^2. \tag{67}$$

Unfolding this inequality then yields

$$\mathbf{E}\mathbf{E}(\widehat{\varphi}^2 \mid \boldsymbol{T}) \leq \prod_{0<i\leq r} \left(1 + \max_{\alpha\in A_i} \bar{\nu}_\alpha\right) \mathbf{E}^2\varphi, \tag{68}$$

which implies the theorem, since

$$\mathbf{E}\mathbf{V}(\widehat{\varphi} \mid \boldsymbol{T}) = \mathbf{E}\mathbf{E}(\widehat{\varphi}^2 \mid \boldsymbol{T}) - \mathbf{E}\mathbf{E}^2(\widehat{\varphi} \mid \boldsymbol{T}) \tag{69}$$

$$\leq \mathbf{E}\mathbf{E}(\widehat{\varphi}^2 \mid \boldsymbol{T}) - \mathbf{E}\varphi^2 \tag{70}$$

$$\leq \mathbf{E}\mathbf{E}(\widehat{\varphi}^2 \mid \boldsymbol{T}) - \mathbf{E}^2\varphi. \tag{71}$$

□

Thus, we can deduce the performance of HS from the "goodness" of h, as quantified by the parameters $\widetilde{\nu}$ and $\bar{\nu}$.

4.7. A random graph problem. We illustrate the previous theorem with the following example. Consider the maximum independent set problem in which the object is to find the maximum number of independent vertices with each pair not connected by an edge. Suppose that we take the obvious approach of solving this problem by exhaustively enumerating every independent set in the graph [10]. Then it is useful to estimate the size (the number of nodes) of the search tree before running the actual brute-force algorithm. Thus, imagine applying HS with $f = 1$ on the backtrack tree induced by the input graph $G = (V, E)$ to the maximum independent set problem. In the backtrack tree, each node s corresponds to an independent set I as well as to the subgraph $G' = (V', E')$ of G with I and its neighborhood removed. Since our objective is to estimate the size of the tree, it is both natural and simple to use the cardinality of V' as a stratifier.

Now suppose that the input graph G is random in that every edge is independently likely to occur with probability $p = 1/2$. We can proceed to examine the average variance of the estimator produced by HS by obtaining some bound on $\bar{\nu}$. Thus, we need to study the distribution of Φ_n, with Φ_n being the size of the random tree induced by a random graph G_n of order n. Since Φ_n is equal to the number of independent sets in G_n, we can use standard techniques in random graph theory to deduce that

$$\mathbf{V}\Phi_n = O\left(\frac{\lg^4 n}{n^2}\right) \mathbf{E}^2\Phi_n, \tag{72}$$

where $\lg n$ denotes the binary logarithm of n. (A proof is provided in the appendix.) Therefore, we can immediately conclude the following.

THEOREM 15. *Suppose, as described in the beginning of this section, that we apply* HS *with the stratifier that uses the cardinality V', on the backtrack tree induced by a random graph of order n, with edge probability $p = 1/2$. Then the variance of the estimator $\widehat{\varphi}$, averaged over the possible random trees, is only on the order of the square of the average size of the tree. That is, we have*

$$\mathbf{E}(\mathbf{V}\widehat{\varphi} \mid \boldsymbol{T}) = O(\mathbf{E}^2\varphi). \tag{73}$$

Proof. Combining Theorem 14 with (72), we have for some positive constant c that

$$\begin{aligned} \mathbf{E}(\mathbf{V}\widehat{\varphi} \mid \boldsymbol{T}) &\le \left(\prod_{1\le k<n}\left(1 + c\frac{\lg^4 k}{k^2}\right) - 1\right)\mathbf{E}^2\varphi \\ &< \left(\prod_{k=1}^{\infty}\left(1 + c\frac{\lg^4 k}{k^2}\right) - 1\right)\mathbf{E}^2\varphi, \end{aligned}$$

where the infinite product is convergent. □

Thus, we have demonstrated one situation in which the average performance of HS using a natural stratifier is quite satisfactory, at least in a theoretical sense.

5. Computational experience. The practicality of HS has been tested in a variety of situations [3], so we will describe only one experiment here. Consider the chessboard recreation whose goal is to find the longest uncrossed knight's tour possible on the board (Fig. 2). Obviously, we can try to solve this problem by means of backtracking; however, it is not clear whether an exhaustive approach is feasible. In the original paper [5] where this example appears, Knuth answers the feasibility question by applying

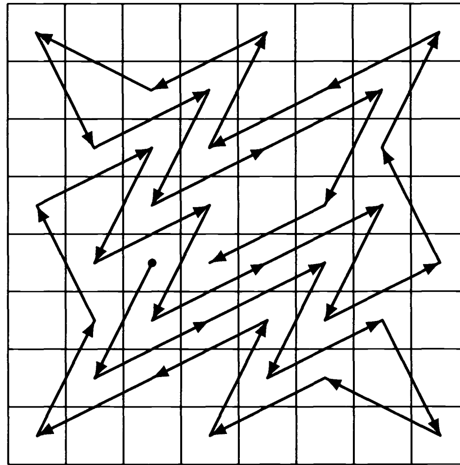


FIG. 2. *A (longest) uncrossed knight's tour.*

SS on the backtrack tree; using just the depth stratifier, he is able to pinpoint the number of nodes in the tree with surprising accuracy.

Although Knuth's experiment is successful in estimating the size of the tree, the estimation results can still be improved significantly by incorporating problem-specific heuristics into the random search algorithm. To illustrate this point, we identify each node in the backtrack tree with a corresponding uncrossed knight's tour, and consider the knight's tour problem as a directed-graph problem in which each vertex corresponds to a square on the chessboard, and each edge corresponds to a legal knight's move. Each edge of course precludes a number of other edges because we are only interested in tours that do not cross themselves. Now, among the knight's tours, we expect those tours that corner themselves to have less "mobility," and hence less possibility for larger subtrees than those that do not. We can gauge the mobility of a tour by counting the number of edges that have not yet been precluded by the tour and can still be reached with the edge preclusion condition relaxed. The resulting function, denoted by mobility(s) with s being the node corresponding to the tour, is in fact a stratifier, as can be verified. With h_0 being the simple depth stratifier, and h_1 being the mobility function just defined, we can construct yet another stratifier h_2 by simply concatenating the values of the two former ones.

The respective performance of HS using these three stratifiers is shown empirically in Table 1. First, we repeated Knuth's experiment [5] by recording the mean number of nodes predicted for each level after running the h_0 algorithm 1000 times. (The data obtained do not coincide with the original ones because our pseudo-random number generators were different.) Next, we ran the h_1 algorithm 40 times, the h_2 algorithm 5 times, and recorded the corresponding estimates. For simplicity, we measured the total cost of each algorithm by the number of nodes examined. This accounting explains the numbers 40 and 5; for that was when the latter two algorithms had examined roughly the same number of nodes as the first one. To serve as a reference, the true profile of the tree is reproduced here. Incidentally, for a fairer comparison, we should really take into account the cost of evaluating mobility(s). Because the h_0 algorithm need not compute this function, we could probably run it 6000 times to equal one trial of sampling with h_2.

TABLE 1
Estimated tree profiles.

Level	h_0	h_1	h_2	Actual
0	1	1	1	1
1	10	10	10	10
2	43	42	34	42
3	257	196	204	251
4	1005	918	864	968
5	4267	3734	3604	4215
6	15684	14026	13724	15646
7	57443	46873	51326	56435
8	184017	126007	174680	182520
9	583197	445188	611950	574555
10	1608183	1613621	1832792	1606422
11	4371678	4679457	5106228	4376153
12	9920481	16793389	11767976	10396422
13	22507886	37750158	25507044	23978392
14	47054083	31161416	47574616	47667686
15	95846027	64365367	83428348	91377173
16	156085563	106268997	130208416	150084206
17	256650972	205655280	192462996	235901901
18	344261889	306859328	252526926	315123658
19	451703678	382647057	280613374	399772215
20	324806815	372852597	277536920	427209856
21	253992102	426672714	272541294	429189112
22	237855139	239446386	288235830	358868304
23	106973568	352833124	226878632	278831518
24	80018150	195532658	138579650	177916192
25	28304640	57001796	87525874	103894319
26	0	72444273	40045814	49302574
27	0	19732265	16732876	21049968
28	0	151248453	9984256	7153880
29	0	0	2985372	2129212
30	0	0	787150	522186
31	0	0	138532	109254
32	0	0	13210	18862
33	0	0	588	2710
34	0	0	588	346
35	0	0	0	50
36	0	0	0	8
total	2422806779	3046195329	2393871699	3137317290
std dev	10672087188	2331712470	878149488	
cost	11503	10875	11345	
trials	1000	40	5	

However, this comparison would depend more on the actual implementation.

It is apparent that the estimation power of HS increases upon the application of more sophisticated stratifiers. True, the three estimates for the actual size of the tree are all accurate enough to give us the right order of magnitude, but the estimates produced by HS using h_1 or h_2 are much more stable and hence give us more confidence than those produced by the algorithm using h_0. This stability is indicated by the decreasing order of magnitude of the observed deviation as we go from h_0 to h_2.

Besides estimating the size of the tree, we can also use HS to estimate the entire profile of the tree. By computing the product of the branching degrees of path associated with one of the eight 35-move solution, we reckon that it would take about a billion trial

runs for SS to come up with a nonzero estimate for the number of nodes at the bottom level. Fortunately, as Table 1 clearly shows, the sampling technique can probe deeper into the tree with the aid of simple heuristics. Take, for example, the stratifier h_2; in just five trial runs, it has enabled HS to find a 33-move knight's tour.

In addition to estimating the cost of brute-force backtracking, we can also predict the performance of other heuristic search algorithms by profiling the backtrack tree according to the heuristic used. For example, consider the heuristic $F(s)$, which when given a state s with a partial uncrossed knight's tour, computes the total number of vertices whose set includes those already on the tour and those still reachable with the edge-preclusion condition relaxed. Because F is monotonically nonincreasing, we can invoke a best-first search strategy and be certain of an optimum solution when a leaf node is reached. However, a best-first strategy may require too much space; in this case, an iterative-deepening approach may be more appropriate. Using HS, we can predict whether these heuristic search algorithms are indeed better than simple backtracking by first estimating

$$\varphi_1(F_0) = \sum_s \lceil F(s) = F_0 \rfloor, \tag{74}$$

where $\lceil \cdot \rfloor$ denotes the indicator function: 1 if the predicate is satisfied, and zero otherwise. Then, by calculating

$$\varphi_2(F_0) = \sum_{F_1 \geq F_0} \varphi_1(F_1), \tag{75}$$

we can predict the number of nodes that a best-first search has to expand before it can process any s with $F(s) < F_0$. Likewise, by further summing

$$\varphi_3(F_0) = \sum_{F_1 \geq F_0} \varphi_2(F_1), \tag{76}$$

we can predict the number of nodes that an iterative-deepening search (with ∞ as the initial bound) has to expand before it can process any s with $F(s) < F_0$.

Shown in Table 2 is the result of applying HS 10 times using stratifier h_2. For simplicity, we list only the average of the estimates for $F_0 \geq 30$, and compare only those with $F_0 \geq 36$ with the actual values that were computed later. The estimates are all quite accurate in determining the order of magnitude of the actual values, even though the total number of nodes examined by HS (22750) is only about $1/10000$ of that required by an exhaustive search. In our implementation, the sampling procedure took about 10 minutes on a Sun, while the exact computation took about 3 days of work distributed among 4 Suns.

Incidentally, if we know that an optimum tour has length around 35, then with the estimates we would be able to predict the infeasibility of a best-first search and the inferiority of an iterative-deepening approach with respect to simple backtracking. A best-first search would require us to maintain a queue of about 10^8 nodes, even though it needs only to examine about $1/15$ of the nodes in the entire backtrack tree. On the other hand, an iterative-deepening approach with ∞ as the initial bound would require an examination of about $1/3$ of the backtrack tree, a factor which is not small enough to offset the cost of evaluating F. However, if we know that the optimum length is definitely equal to 35, then an iterative-deepening approach with 36 as the initial bound on F might now be better than simple backtracking, since it needs to examine only about $1/15$ of the tree.

TABLE 2
Tree profile under a monotone heuristic.

F_0	$\widehat{\varphi}_1(F_0)$	$\varphi_1(F_0)$	$\widehat{\varphi}_2(F_0)$	$\varphi_2(F_0)$	$\widehat{\varphi}_3(F_0)$	$\varphi_3(F_0)$
64	39815	39379	39815	39379	39815	39379
63	111703	86641	151518	126020	191333	165399
62	144451	151599	295969	277619	487302	443018
61	226524	228636	522493	506255	1009795	949273
60	433814	309611	956307	815866	1966102	1765139
59	427256	403924	1383563	1219790	3349665	2984929
58	530568	512112	1914131	1731902	5263796	4716831
57	845676	614624	2759807	2346526	8023603	7063357
56	825869	756939	3585676	3103465	11609279	10166822
55	1329818	923511	4915494	4026976	16524773	14193798
54	1228600	1139180	6144094	5166156	22668867	19359954
53	991948	1439727	7136042	6605883	29804909	25965837
52	2659514	1793783	9795556	8399666	39600465	34365503
51	2669262	2211864	12464818	10611530	52065283	44977033
50	2499762	2741692	14964580	13353222	67029863	58330255
49	4332987	3379471	19297567	16732693	86327430	75062948
48	3465842	4178690	22763409	20911383	109090839	95974331
47	5277864	5144058	28041273	26055441	137132112	122029772
46	5980861	6322974	34022134	32378415	171154246	154408187
45	7613390	7753753	41635524	40132168	212789770	194540355
44	9427006	9486538	51062530	49618706	263852300	244159061
43	10839121	11483764	61901651	61102470	325753951	305261531
42	15375928	13812502	77277579	74914972	403031530	380176503
41	13808601	16432504	91086180	91347476	494117710	471523979
40	15494192	19418400	106580372	110765876	600698082	582289855
39	24607447	22642483	131187819	133408359	731885901	715698214
38	21976276	26231844	153164095	159640203	885049996	875338417
37	29037331	29968602	182201426	189608805	1067251422	1064947222
36	31250938	33920456	213452364	223529261	1280703786	1288476483
35	29411721		242864085		1523567871	
34	35719965		278584050		1802151921	
33	37585184		316169234		2118321155	
32	40970294		357139528		2475460683	
31	34998800		392138328		2867599011	
30	49462557		441600885		3309199896	

6. Conclusions. Heuristic sampling is a simple, easy to apply, and easy to implement technique designed to predict the performance of other tree searching programs. The flexibility and practicality of this technique stem from our accessibility to the input stratifier, through which natural heuristics can be introduced to guide and control the estimation process. In evaluating its effectiveness, we have shown the unbiasedness of the estimator and investigated the conditions for minimum variance. We have also demonstrated a relationship between the quality of the stratifier and the accuracy of the estimator by formalizing the concept of stratum homogeneity. In supporting our general results, we have also conducted some specific case analysis, one involving theoretical computation, and the other, experimental simulation. The results obtained are all very encouraging. Overall, we expect heuristic sampling to be an invaluable asset in aiding the design of tree searching programs as we use the technique to compare the merits of various search algorithms.

Appendix. Let G_n be a random graph with n vertices and independent edge probability p, and let Φ_n be the number of independent sets in G_n. To study the expectation

and the variance of Φ_n, we use related techniques and results derived in §4 of [10], §XI.1 of [1], and §5.3 of [7]. Thus, imagine an enumeration of all 2^n subgraphs H_i of G_n. Let X_i be the 0-1 random variable indicating the independence of H_i; we have $\Phi_n = \sum_i X_i$. The probability that $X_i = 1$, or the probability that the subgraph H_i is independent, is equal to $q^{\binom{k}{2}}$, where k is the order of this subgraph and $q = 1 - p$. Since there are $\binom{n}{k}$ subgraphs of order k, we have

$$\mathbf{E}\Phi_n = \sum_k \binom{n}{k} q^{\binom{k}{2}}, \tag{77}$$

and according to Wilf, for $p = q = 1/2$, this sum has most of its contributions coming from terms with k around $\lg n$.

To compute $\mathbf{V}\Phi_n$, let us first consider

$$\mathbf{E}\Phi_n^2 = \sum_{i,j} \mathbf{E}X_iX_j. \tag{78}$$

Suppose that the subgraphs H_i and H_j are of orders k and l, respectively. Then the probability that $X_iX_j = 1$, or the probability that both H_i and H_j are independent, is equal to $q^{\binom{k}{2}+\binom{l}{2}-\binom{r}{2}}$, where r is the number of vertices H_i and H_j have in common. On the other hand, $\binom{n}{k-r,l-r,r}$ is the number of such pairs of subgraphs having r vertices in common, with the first of order k and the second of order l. Hence, we have

$$\mathbf{E}\Phi_n^2 = \sum_{k,l,r} \binom{n}{k-r, l-r, r} q^{\binom{k}{2}+\binom{l}{2}-\binom{r}{2}}, \tag{79}$$

and under the same interpretation, we have

$$\mathbf{E}^2\Phi_n = \sum_{k,l,r} \binom{n}{k-r, l-r, r} q^{\binom{k}{2}+\binom{l}{2}}, \tag{80}$$

which gives us

$$\mathbf{V}\Phi_n = \sum_{k,l} q^{\binom{k}{2}+\binom{l}{2}} \sum_{2\le r\le \min(k,l)} \binom{n}{k-r, l-r, r} (q^{-\binom{r}{2}} - 1). \tag{81}$$

Thus, we define $b = 1/q$ and analyze the quantity

$$A_{k,l} \stackrel{\text{def}}{=} q^{\binom{k}{2}+\binom{l}{2}} \sum_{2\le r\le k} \binom{n}{k-r, l-r, r} b^{\binom{r}{2}} \tag{82}$$

for different regions of k and l to bound $\mathbf{V}\Phi_n$.

LEMMA 16. *If $k \le l$ and $l \ge 1 + 4\log_b n$, then*

$$A_{k,l} \le 1. \tag{83}$$

Proof.

$$A_{k,l} \le q^{\binom{l}{2}} \sum_{r\le k} \binom{n}{k-r, l-r, r} \tag{84}$$

$$= q^{\binom{l}{2}} \binom{n}{k} \binom{n}{l} \tag{85}$$

$$\leq \left(\frac{n^2}{b^{(l-1)/2}} \right)^l \tag{86}$$

$$\leq 1^l = 1. \tag{87}$$

□

LEMMA 17. *Let*

$$B_r \stackrel{\text{def}}{=} \binom{n}{k-r, l-r, r} b^{\binom{r}{2}}. \tag{88}$$

If $l < 5 \log_b n$ *and* $3 \leq r \leq k \leq l$, *then*

$$B_r < B_3 + B_k \tag{89}$$

for sufficiently large n.

Proof. Suppose that $r \leq \frac{3}{2} \log_b n$. Then consider the ratio

$$a_r \stackrel{\text{def}}{=} B_r / B_3. \tag{90}$$

It is equal to

$$\left(\frac{3!}{r!} \right) \left(\frac{(k-3)!}{(k-r)!} \right) \left(\frac{(l-3)!}{(l-r)!} \right) \left(\frac{(n-k+3)!}{(n-k+r)!} \right) b^{\binom{r}{2}-3}, \tag{91}$$

which is bounded above by

$$\left(\frac{(k-3)(l-3)}{(n-k+3)} \right)^{r-3} b^{r(r-1)/2}. \tag{92}$$

Hence, for sufficiently large n, we have

$$\begin{aligned} a_r &= O(klb^{r/2}/n)^r && (93) \\ &= O(kln^{3/4}/n)^r && (94) \\ &= o(1). && (95) \end{aligned}$$

Now suppose that $r \geq \frac{3}{2} \log_b n$. Then consider the ratio

$$b_r \stackrel{\text{def}}{=} B_r / B_k. \tag{96}$$

After expansion, it becomes

$$\left(\frac{0!}{(k-r)!} \right) \left(\frac{k!}{r!} \right) \left(\frac{(l-k)!}{(l-r)!} \right) \left(\frac{(n-l)!}{(n-l-(k-r))!} \right) b^{\binom{r}{2}-\binom{k}{2}}, \tag{97}$$

which can be bounded above by

$$(kn)^{k-r} / b^{\binom{k}{2}-\binom{r}{2}}. \tag{98}$$

But notice that $k \geq r$ implies that

$$\binom{k}{2} - \binom{r}{2} \leq \frac{r(k-1)}{2} - \frac{r(r-1)}{2} = \frac{r(k-r)}{2}. \tag{99}$$

Hence, for sufficiently large n, we have

$$b_r = O(kn/b^{r/2})^{k-r} \tag{100}$$
$$\leq O(kn/n^{3/2})^{k-r} \tag{101}$$
$$= o(1). \tag{102}$$

□

LEMMA 18. *If $k \leq l < 5\log_b n$, then for sufficiently large n, we have*

$$A_{k,j} \leq \left(\frac{bl^4}{n^2} + \frac{b^3 l^7}{n^3}\right) E_k E_l + l^{l+1} E_l, \tag{103}$$

where

$$E_k \stackrel{\text{def}}{=} \binom{n}{k} q^{\binom{k}{2}}. \tag{104}$$

Proof. According to the previous lemma, we can bound $A_{k,j}$ by

$$q^{\binom{k}{2}+\binom{l}{2}}(B_2 + (k-2)(B_3 + B_k)). \tag{105}$$

Next, observe that

$$\binom{n}{l-k, 0, k} = \binom{n}{l}\binom{l}{k} \leq \binom{n}{l} l^l \tag{106}$$

and

$$\binom{n}{k-r, l-r, r} \leq \frac{k^r l^r}{r!\,(n-r)^r}\binom{n}{k}\binom{n}{l} \tag{107}$$
$$\leq \frac{k^r l^r}{n^r}\binom{n}{k}\binom{n}{l}. \tag{108}$$

The lemma then follows immediately by substituting these bounds into the definition of B_r. □

LEMMA 19. *For sufficiently large n, and $p = q = 1/2$, we have*

$$\mathbf{V}\Phi_n = O\left(\frac{\lg^4 n}{n^2}\right)\mathbf{E}^2\Phi_n. \tag{109}$$

Proof. From Lemma 16 comes the bound

$$\mathbf{V}\Phi_n \leq \sum_{k,l} 1 + \sum_{k,l<5\lg n} A_{k,l}. \tag{110}$$

Next, we apply Lemma 18 to bound $A_{k,l}$. By setting $p = q = 1/2$, we have

$$\mathbf{V}\Phi_n \leq n^2 + O\left(\frac{\lg^4 n}{n^2}\right)\sum_{k,l} E_k E_l + n^{O(\lg\lg n)}\sum_k E_k, \tag{111}$$

which implies the result since

$$\textstyle\sum_r E_r = \mathbf{E}\Phi_n = \Omega(n^{\frac{1}{2}\lg n - \lg\lg n}). \tag{112}$$

□

Acknowledgment. This paper is based on the author's thesis supervised by Don Knuth. The author thanks Ernie Brickell, Carl Diegert, and David Strip for their encouragement and review of this publication.

REFERENCES

[1] B. BOLLOBÁS, *Random Graphs*, Academic Press, New York, 1985.
[2] P. BRATLEY, B. L. FOX, and L. E. SCHRAGE, *A Guide to Simulation*, Second Ed., Springer-Verlag, Berlin, New York, 1987.
[3] P. CHEN, *Heuristic Sampling on Backtrack Trees*, Ph.D. thesis, Stanford University, Stanford, CA, 1989.
[4] S. KARLIN AND H. M. TAYLOR, *A First Course in Stochastic Processes*, Second Ed., Academic Press, New York, 1975.
[5] D. E. KNUTH, *Estimating the efficiency of backtrack programs*, Math. Comp., 29 (1975), pp. 121–136.
[6] R. E. KORF, *Depth-first iterative-deepening: An optimal admissible tree search*, Artificial Intelligence, 27 (1985), pp. 97–109.
[7] E. M. PALMER, *Graphical Evolution*, John Wiley, New York, 1985.
[8] J. PEARL, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Boston, MA, 1984.
[9] P. W. PURDOM, *Tree size by partial backtracking*, SIAM J. Comput., 7 (1978), pp. 481–491.
[10] H. S. WILF, *Some examples of combinatorial averaging*, Amer. Math. Monthly, 92 (1985), pp. 250–261.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 316–328, April 1992
© 1992 Society for Industrial and Applied Mathematics
008

COUNTING CLASSES ARE AT LEAST AS HARD AS THE POLYNOMIAL-TIME HIERARCHY*

SEINOSUKE TODA† AND MITSUNORI OGIWARA†

Abstract. In this paper, it is shown that many natural counting classes, such as PP, $C_{=}P$, and MOD_kP, are at least as computationally hard as PH (the polynomial-time hierarchy) in the following sense: for each **K** of the counting classes above, every set in **K**(PH) is polynomial-time randomized many-one reducible to a set in **K** with two-sided exponentially small error probability. As a consequence of the result, it is seen that all the counting classes above are computationally harder than PH unless PH collapses to a finite level. Some other consequences are also shown.

Key words. counting complexity classes, polynomial-time hierarchy, randomized reducibility, computational complexity theory

AMS(MOS) subject classifications. 68Q15, 03D15

1. Introduction.

In the theory of computational complexity, researchers have given much attention to several questions about the computational power of counting complexity classes such as PP defined by Gill [10], $C_{=}P$ defined by Wagner [26], $\oplus P$ defined by Papadimitriou and Zachos [15], and MOD_kP defined independently by Cai and Hemachandra [7] and by Beigel, Gill, and Hertrampf [4]. In those investigations, it is of particular interest to compare the computational power of the counting classes with that of classes within PH (the polynomial-time hierarchy), and the researchers have considered two different types of questions, containment questions and reducibility questions. By a containment question, we mean to ask whether a class in PH is included in a counting class, and by a reducibility question, we mean to ask whether all sets in a class in PH are polynomial-time reducible to sets in a counting class under a suitable reducibility.

For the containment question on PP, the best result at the present time is one by Beigel, Hemachandra, and Wechsung [3] that $P^{NP[\log]}$ is included in PP. For other counting classes, it was shown in [4], [7], and [13] that Few $\subseteq \oplus P$, Few $\subseteq C_{=}P$, and for each prime k, Few $\subseteq MOD_kP$, where Few was defined by Cai and Hemachandra [7] (see their paper for the detail) and is known to be below $P^{NP[\log]}$. These are the best results for the containment questions on all the known counting classes other than PP. Very recently, Fenner, Fortnow, and Kurtz [8] have unified and improved the results on Few versus counting classes questions. The reader may refer to [8] for the current status of these results.

For some reducibility questions, it was shown by Toda [21] that all sets in PH is polynomial-time Turing reducible to sets in PP and are polynomial-time randomized reducible to sets in $\oplus P$ (here and throughout the paper all randomized reductions are of two-sided exponentially small error probability). It was recently shown by Ogiwara [14] that all sets in Π_2^P are polynomial-time randomized many-one reducible to sets in $C_{=}P$. Combining these results with a result by Schöning [17], we can conclude that PP and $\oplus P$ are computationally harder than PH unless PH collapses to a finite level, and conclude that $C_{=}P$ are computationally harder than Σ_2^P unless PH collapses to a finite level.

*Received by the editors July 14, 1990; accepted for publication (in revised form) April 22, 1991. This research was done while the authors visited the Department of Mathematics, University of California, Santa Barbara, California 93106, and it was supported in part by National Science Foundation grant CCR89-13584. Portions of this work were presented at the 6th IEEE Conference on Structure in Complexity Theory (which was held from June 30 until July 3, 1991).

†Department of Computer Science and Information Mathematics, University of Electro-Communications, Chofu-shi, Tokyo 182, Japan.

While several relationships between the counting classes and classes in PH have been established, many relationships are currently unknown. In fact, we do not know at present whether PP includes Δ_2^{P} and whether the other counting classes include NP. Nor did we know whether all sets in PH are polynomial-time reducible to sets in the counting classes other than PP and $\oplus$P. Recently some oracle sets refuting the expected containments have been found. Toran [22] found oracle sets A and B such that NP(A) $\not\subseteq$ $\mathrm{C_{=}P}(A)$ and NP(B) $\not\subseteq$ $\oplus$P(B), Beigel [2] found an oracle set C such that $\Delta_2^{\mathrm{P}}(C)$ $\not\subseteq$ PP(C), and he showed in [1] that for all $k \geq 2$ and some oracle set D, NP(D) $\not\subseteq$ $\mathrm{MOD}_k\mathrm{P}(D)$. These relativization results tell us at least that all the techniques known currently do not work for settling the containment questions above, and/or that it is too difficult to solve the questions. Hence the important question at present is whether all sets in PH are polynomial-time reducible to sets in those counting classes under a suitable reducibility.

In this paper we investigate the reducibility question above more deeply. We will be concerned with the polynomial-time randomized many-one reducibility with two-sided exponentially small error probability (this notion will be formalized as the $\widehat{\mathrm{BP}}$-operator in §2). We will conclude that all sets in PH are polynomial-time randomized many-one reducible to sets in a wide range of counting classes. To show this, we will first introduce a family of counting classes that is a restricted range of the gap-definable classes recently developed by Fenner, Fortnow, and Kurtz [8] but is still so wide as to include PP, $\mathrm{C_{=}P}$, and $\mathrm{MOD}_k\mathrm{P}$, and we will show that all counting classes in the family are at least as hard as PH. Thus, in particular, we establish the following new relationships as immediate consequences of the general result (we also summarize some relationships known earlier for the sake of comparison of our results with them):

(1) PP(PH) $\subseteq$ P(PP) and $\oplus$P(PH) $\subseteq$ $\widehat{\mathrm{BP}} \cdot \oplus\mathrm{P}$ [21].

(2) $\Pi_2^{\mathrm{P}} \subseteq \widehat{\mathrm{BP}} \cdot \mathrm{C_{=}P}$ [14]. Here we note that the inclusion $\Sigma_2^{\mathrm{P}} \cup \Pi_2^{\mathrm{P}} \subseteq \widehat{\mathrm{BP}}\cdot\mathrm{PP}$ immediately follows from this result.

(3) $\mathrm{C_{=}P(PH)} \subseteq \widehat{\mathrm{BP}} \cdot \mathrm{C_{=}P}$ and $\mathrm{PP(PH)} \subseteq \widehat{\mathrm{BP}}\cdot\mathrm{PP}$. These improve the relationships in (2) above.

(4) For all integers $k \geq 2$, $\mathrm{MOD}_k\mathrm{P(PH)} \subseteq \widehat{\mathrm{BP}}\cdot\mathrm{MOD}_k\mathrm{P}$. This extends the second result in (1) above.

As an immediate consequence, we know that PH is included in $\widehat{\mathrm{BP}}\cdot\mathbf{K}$ for each $\mathbf{K} \in \{$ PP, $\mathrm{C_{=}P}$, $\mathrm{MOD}_k\mathrm{P}$ $\}$.

Remark. After seeing an earlier version of the present paper, Richard Beigel told us as a private communication that the result (4) could be obtained by using the result in [4, Thm. 27] that $\mathrm{MOD}_k\mathrm{P}(\mathrm{MOD}_k\mathrm{P}) = \mathrm{MOD}_k\mathrm{P}$ if k is a prime, the characterization of $\mathrm{MOD}_k\mathrm{P}$ in [4, Cor. 33] (every MOD-class is a finite union of some MOD-classes with prime modulo), and the modulo amplification technique in [21]. It is important to remark that Tarui [19] has independently developed the same techniques as those in this paper, and in fact he observed somewhat stronger relationships than ours. Roughly speaking, he has strengthened the relationships in (3) and (4) above via the polynomial-time randomized reducibility with *zero error probability*.

This paper will proceed as follows. In §2, we first state our main result without giving the main part of the technical proof, and show its immediate consequences. The main part of the proof will be given in §3. In the rest of this section, we give some elementary notions and notations used throughout the paper.

Our sets in this paper are over $\Sigma = \{0, 1, \#\}$ unless otherwise specified. The symbol $\#$ is usually used as a delimiter among strings of $\{0, 1\}^*$. A pairing function (respectively, a k-tuple function) over $\{0, 1\}^*$ is represented by separating two strings (respectively, k

strings) by this symbol. For a string $w \in \Sigma^*$, $|w|$ denotes the length of w. For a set $L \subseteq \Sigma^*$, $\overline{L}$ denotes the complement of L. For a class **K** of sets, co-**K** denotes the class of sets whose complement is in **K**. Let Σ^n (respectively, $\Sigma^{\leq n}$ and $\Sigma^{<n}$) denote the set of strings with length n (respectively, at most n and less than n). For a finite set $X \subseteq \Sigma^*$, $\|X\|$ denotes the number of strings in X. Let **N** and **Z**, respectively, denote the set of natural numbers and the set of integers. We assume that all integers are expressed in binary notation (for negative integers, we assume a suitable representation system, such as 2's-complement system).

For an oracle set X, P(X) denotes the class of sets accepted by polynomial-time bounded deterministic oracle Turing machines (DOTM) with oracle X, and NP(X) denotes the class of sets accepted by polynomial-time bounded nondeterministic oracle Turing machines (NOTM) with oracle set X. Classes in the polynomial-time hierarchy [18] are denoted in the usual way : $\Sigma_0^{\mathrm{P}} = \Pi_0^{\mathrm{P}} = \Delta_0^{\mathrm{P}} = \mathrm{P}$, $\Sigma_k^{\mathrm{P}} = \mathrm{NP}(\Sigma_{k-1}^{\mathrm{P}})$, $\Pi_k^{\mathrm{P}} = \text{co-}\Sigma_k^{\mathrm{P}}$, $\Delta_k^{\mathrm{P}} = \mathrm{P}(\Sigma_{k-1}^{\mathrm{P}})$, and $\mathrm{PH} = \bigcup_{k\geq 0} \Sigma_k^{\mathrm{P}}$.

We are concerned with the following counting classes. #P(X) [24] denotes the class of functions that give the number of accepting computation paths of polynomial-time bounded NOTM's with oracle X. PP(X) [10] (respectively, $\mathrm{C_=P}(X)$ [26]) denotes the class of sets L for which there exist two functions $F_1, F_2 \in$ #P(X) such that for all strings x, $x \in L$ if and only if $F_1(x) > F_2(x)$ (respectively, $x \in L$ if and only if $F_1(x) = F_2(x)$). For an integer $k \geq 2$, we define $\mathrm{MOD}_k\mathrm{P}(X)$ [4] as the class of sets L for which there exists a function $F \in$ #P(X) such that for all strings x, $x \in L$ if and only if $F(x) \not\equiv 0 \pmod{k}$. In particular, $\mathrm{MOD_2P}$ is usually denoted by $\oplus$P [15]. The unrelativized classes are defined by setting the oracle set to the empty set, and the specification of the oracle is omitted in this case.

2. Randomized reductions from PH to counting classes. In this section we show that all sets in PH are polynomial-time randomized many-one reducible to sets in a wide range of counting classes, including PP, $\mathrm{C_=P}$, and $\mathrm{MOD}_k\mathrm{P}$. What we will show is much stronger than this observation. For example, all sets in PP(PH) are polynomial-time randomized many-one reducible to sets in PP. More precisely, we first define a stronger variation of Schöning's BP-operator [17], which formalizes the notion of polynomial-time randomized many-one reducibility with two-sided *exponentially small* error probability, and next define a family of counting classes to which our technique can be applied. Below, given a finite set X of strings and a predicate R over strings, we denote by $\Pr\{w \in X : R(w)\}$ the probability that $R(w)$ is true for randomly chosen w from X under uniform distribution.

DEFINITION 2.1. Let **K** be any class of sets. A set L is in $\widehat{\mathrm{BP}}\cdot\mathbf{K}$ if for every polynomial e, there exist a set $A \in \mathbf{K}$ and a polynomial p such that for every string x,

$$\Pr\{w \in \{0,1\}^{p(|x|)} : x \in L \text{ iff } x\#w \in A\} \geq 1 - 2^{-e(|x|)}.$$

The essence of the following definition comes from the gap-definability notion of Fenner, Fortnow, and Kurtz [8], and our notion below covers a wide range of counting classes while it defines a smaller family of their gap-definable classes.

DEFINITION 2.2. For an oracle set X, GapP(X) [8] is defined to be the class of functions F for which there exist two functions $F_1, F_2 \in$ #P(X) such that for all strings x, $F(x) = F_1(x) - F_2(x)$, and GapP is defined as GapP($\emptyset$). Let Q be a subset of **Z** and let $\mathcal{F}$ be a class of functions from strings to integers. Then we define $\mathcal{C}[Q, \mathcal{F}]$ to be the class of sets L for which there exists a function $F \in \mathcal{F}$ such that for all x, $x \in L$ if and only if $F(x) \in Q$.

Example. All the well-known counting classes can be defined as in Definition 2.2. For instance, PP and PP(X), for any oracle set X, can be defined as $\mathcal{C}[\mathbf{Z}^+, \text{GapP}]$ and $\mathcal{C}[\mathbf{Z}^+, \text{GapP}(X)]$, respectively, where $\mathbf{Z}^+$ denotes the set of positive integers.

Our main result essentially follows from the following technical lemma, whose proof will be given in the next section.

LEMMA 2.3. *Let F be any function in* GapP(PH) *and let e be any polynomial. Then there exist a function $H \in$* GapP *and a polynomial s such that for every string x,*

$$\Pr\{w \in \{0,1\}^{s(|x|)} : H(x\#w) = F(x)\} \geq 1 - 2^{-e(|x|)}.$$

Our main theorem below subsumes disparate observations that several counting classes are at least as computationally hard as PH.

THEOREM 2.4. *Let Q be a subset of $\mathbf{Z}$. Then,* $\mathcal{C}[Q, \text{GapP(PH)}] \subseteq \widehat{\text{BP}} \cdot \mathcal{C}[Q, \text{GapP}]$.

Proof. Let F be a function in GapP(PH) witnessing that a set L is in $\mathcal{C}[Q, \text{GapP(PH)}]$. From Lemma 2.3, there exists a function $H \in$ GapP that satisfies the condition in the lemma. Define $A = \{\, x\#w \mid H(x\#w) \in Q \,\}$. Obviously, A is in $\mathcal{C}[Q, \text{GapP}]$, and L and A satisfy the condition in Definition 2.1. Thus we have $L \in \widehat{\text{BP}} \cdot \mathcal{C}[Q, \text{GapP}]$. □

In the rest of this section, we apply the theorem to some well-known counting classes and obtain some new relationships between those counting classes and PH. Also, we will show some other interesting consequences of the theorem and the corollary below.

COROLLARY 2.5. (1) PP(PH) $\subseteq \widehat{\text{BP}}\cdot$ PP.

(2) $\text{C}_=\text{P(PH)} \subseteq \widehat{\text{BP}}\cdot \text{C}_=\text{P}$ *and* co-$\text{C}_=\text{P(PH)} \subseteq \widehat{\text{BP}}\cdot$ co-$\text{C}_=\text{P}$.

(3) *For all integers $k \geq 2$,* $\text{MOD}_k\text{P(PH)} \subseteq \widehat{\text{BP}}\cdot \text{MOD}_k\text{P}$ *and* co-$\text{MOD}_k\text{P(PH)} \subseteq \widehat{\text{BP}}\cdot$ co-MOD_kP.

(4) *Thus, for each* $\mathbf{K}$ *of* PP, $\text{C}_=\text{P}$, co-$\text{C}_=\text{P}$, MOD_kP, *and* co-MOD_kP, *we have* PH $\subseteq \widehat{\text{BP}}\cdot\mathbf{K}$.

Proof. As in the previous example, we can easily see that all the counting classes above can be defined as in Definition 2.2. We omit the detail here; the interested reader may refer to [8] for some technical points. □

Remark. For a class $\mathbf{K}$ of sets that is closed downward under the polynomial-time majority reducibility, it follows from the probability amplification lemma of Schöning [17] that $\text{BP}\cdot\mathbf{K} = \widehat{\text{BP}}\cdot\mathbf{K}$, where, for two sets A and B, A is *polynomial-time majority reducible* to B if there exists a polynomial-time computable function g such that, for all x, $g(x) = y_1\#\cdots\#y_m$ $(m \geq 1)$ and $x \in A$ if and only if the majority of y_i's are in B. However, if $\mathbf{K}$ is not closed under the reducibility, then the two classes might be different. It is currently known that PP (respectively, $\text{C}_=\text{P}$ and MOD_kP with k prime) are closed downward under the polynomial-time majority reducibility [4], [6], [11] (see [9] also). Thus, for such classes, we may replace the $\widehat{\text{BP}}$-operators in the above corollary by the BP-operators, without weakening the results. Nonetheless Theorem 2.4 and the result (3) above for the unrestricted case might be weakened when the $\widehat{\text{BP}}$-operators are replaced by the BP-operators.

Combining Corollary 2.5(4) with the following result due to Schöning [17], we can observe that $\text{C}_=\text{P}$ and MOD_kP, as well as PP and $\oplus$P, are computationally harder than PH unless PH collapses to a finite level.

THEOREM 2.6 [17]. *For every $k \geq 1$, if* $\Pi_k^{\text{P}} \subseteq \widehat{\text{BP}} \cdot \Sigma_k^{\text{P}}$, *then* PH $= \Sigma_{k+1}^{\text{P}}$.

COROLLARY 2.7. *Let Q be a subset of $\mathbf{Z}$ such that $Q \neq \emptyset$ and $Q \neq \mathbf{Z}$. Then, if* PH $\subseteq \mathcal{C}[Q, \text{GapP}]$, *then* PH *collapses to a finite level. Thus, for each* $\mathbf{K}$ *of* $\text{C}_=\text{P}$ *and* MOD_kP, $\mathbf{K} \not\subseteq$ PH *unless* PH *collapses to a finite level.*

Proof. We first show that $\mathcal{C}[Q, \mathrm{GapP}]$ has a polynomial-time many-one complete set. Let #SAT be the function that, given a Boolean formula, gives the number of satisfying assignments of the formula. It was observed by Valiant [24] (see also [16]) that #SAT $\in$ #P and for all functions F in #P, there exists a polynomial-time computable function f such that for all strings x, $F(x) = \#\mathrm{SAT}(f(x))$. Now define $\mathrm{DifSAT}_Q = \{ \langle \phi_1, \phi_2 \rangle : \#\mathrm{SAT}(\phi_1) - \#\mathrm{SAT}(\phi_2) \in Q \}$. Then we easily see that DifSAT_Q is polynomial-time many-one complete for $\mathcal{C}[Q, \mathrm{GapP}]$ via the function f. Since $Q \neq \emptyset$ and $Q \neq \mathbf{Z}$, it is obvious that $\mathrm{PH} \subseteq \mathcal{C}[Q, \mathrm{GapP(PH)}]$. The first statement of this corollary follows immediately from these facts and Theorems 2.4 and 2.6. The second statement is immediate from the first. □

Below we show some other consequences of Theorem 2.4 and Corollary 2.5. First we show that for each $\mathbf{K}$ of PP, $\mathrm{C_{=}P}$, and $\mathrm{MOD}_k\mathrm{P}$, $\mathbf{K}(\mathrm{PH})$ and $\mathbf{K}$ itself are interreducible to each other under the polynomial-time randomized reducibility. Next we observe that, when we consider a nonuniform version of the classes in the sense of Karp and Lipton [12] and random analogues of those classes in the sense of Bennett and Gill [5], the inclusions in Corollary 2.5 become equalities. To show these results, we first prove some technical lemmas.

LEMMA 2.8. *Let* $\mathbf{K}$ *be any class of sets that is closed downward under the polynomial-time many-one reducibility. Then,* $\widehat{\mathrm{BP}} \cdot \widehat{\mathrm{BP}} \cdot \mathbf{K} = \widehat{\mathrm{BP}} \cdot \mathbf{K}$.

Proof. Since the inclusion $\widehat{\mathrm{BP}} \cdot \mathbf{K} \subseteq \widehat{\mathrm{BP}} \cdot \widehat{\mathrm{BP}} \cdot \mathbf{K}$ is obvious, we show the converse. Let $L \in \widehat{\mathrm{BP}} \cdot \widehat{\mathrm{BP}} \cdot \mathbf{K}$. Then, for any polynomial e_1, there exist a set $A \in \widehat{\mathrm{BP}} \cdot \mathbf{K}$ and a polynomial p such that for each x,

$$\Pr\{w \in \{0,1\}^{p(|x|)} : x\#w \in A \text{ iff } x \in L\} \geq 1 - 2^{-e_1(|x|)}.$$

Furthermore, for any polynomial e_2, there exist a set $B \in \mathbf{K}$ and a polynomial q such that for each y,

$$\Pr\{u \in \{0,1\}^{q(|y|)} : y\#u \in B \text{ iff } y \in A\} \geq 1 - 2^{-e_2(|x|)}.$$

We now define a set C as follows:

$$C = \{x\#wu : |w| = p(|x|),$$
$$|u| = q(|x\#w|) = q(|x| + 1 + p(|x|)), \text{ and } x\#w\#u \in B\}.$$

Since $\mathbf{K}$ is closed under the polynomial-time many-one reducibility and C is polynomial-time many-one reducible to B, we see that C is in $\mathbf{K}$. For any string x, if $x \in L$, then

$$\begin{aligned}
&\Pr\{wu : x\#wu \in C\} \\
&\quad = (\Pr\{w : x\#w \in A\} \times \Pr(\{u : x\#w\#u \in B\} | x\#w \in A)) \\
&\qquad + (\Pr\{w : x\#w \notin A\} \times \Pr(\{u : x\#w\#u \in B\} | x\#w \notin A)) \\
&\quad \geq (1 - 2^{-e_1(|x|)}) \cdot (1 - 2^{-e_2(|x|)}) > 1 - 2^{-e_1(|x|)} - 2^{-e_2(|x|)},
\end{aligned}$$

where w and u are randomly chosen from $\{0,1\}^{p(|x|)}$ and $\{0,1\}^{q(|x|+1+p(|x|))}$, respectively, under uniform distribution, and $\Pr(X|Y)$ denotes the conditional probability of

the event X under the condition Y. Conversely, if $x \notin L$, then

$$\begin{aligned}
&\Pr\{wu : x\#wu \in C\} \\
&\quad = (\Pr\{w : x\#w \in A\} \times \Pr(\{u : x\#w\#u \in B\} | x\#w \in A)) \\
&\qquad + (\Pr\{w : x\#w \notin A\} \times \Pr(\{u : x\#w\#u \in B\} | x\#w \notin A)) \\
&\quad \leq 2^{-e_1(|x|)} \cdot 1 + 1 \cdot 2^{-e_2(|x|)} = 2^{-e_1(|x|)} + 2^{-e_2(|x|)}.
\end{aligned}$$

Thus, for every x,

$$\Pr\{v \in \{0,1\}^{p(|x|)+q(|x|+1+p(|x|))} : x\#v \in C \text{ iff } x \in L\} \geq 1 - 2^{-e_1(|x|)} - 2^{-e_2(|x|)}.$$

Given any polynomial e, we may take the polynomials e_1 and e_2 so that for all natural numbers n, $2^{-e(n)} > 2^{-e_1(n)} + 2^{-e_2(n)}$. This implies $L \in \widehat{\mathrm{BP}}\cdot\mathbf{K}$. □

Given a class $\mathbf{K}$ of sets, we denote by $\leq_m^P \cdot\mathbf{K}$ the class of sets that are polynomial-time many-one reducible to sets in $\mathbf{K}$. Note that for any class $\mathbf{K}$, $\leq_m^P \cdot \leq_m^P \cdot\mathbf{K} = \leq_m^P \cdot\mathbf{K}$, i.e., $\leq_m^P \cdot\mathbf{K}$ is closed downward under the polynomial-time many-one reducibility, because of the transitivity of the reducibility.

COROLLARY 2.9. *Let Q be a subset of $\mathbf{Z}$. Then,* $\widehat{\mathrm{BP}}\cdot \leq_m^P \cdot\mathcal{C}[Q, \mathrm{GapP(PH)}] = \widehat{\mathrm{BP}}\cdot \leq_m^P \cdot\mathcal{C}[Q, \mathrm{GapP}]$.

Proof. We can easily see that $\leq_m^P \cdot\widehat{\mathrm{BP}} \cdot \mathcal{C}[Q, \mathrm{GapP}] \subseteq \widehat{\mathrm{BP}}\cdot \leq_m^P \cdot\mathcal{C}[Q, \mathrm{GapP}]$. Thus, we have, from Theorem 2.4 and Lemma 2.8, that $\widehat{\mathrm{BP}}\cdot \leq_m^P \cdot\mathcal{C}[Q, \mathrm{GapP(PH)}] \subseteq \widehat{\mathrm{BP}}\cdot \leq_m^P \cdot\widehat{\mathrm{BP}} \cdot \mathcal{C}[Q, \mathrm{GapP}] \subseteq \widehat{\mathrm{BP}} \cdot \widehat{\mathrm{BP}}\cdot \leq_m^P \cdot\mathcal{C}[Q, \mathrm{GapP}] = \widehat{\mathrm{BP}}\cdot \leq_m^P \cdot\mathcal{C}[Q, \mathrm{GapP}]$. The converse inclusion is obvious. □

The following corollary is immediate from the one above.

COROLLARY 2.10. (1) $\widehat{\mathrm{BP}}\cdot \mathrm{PP(PH)} = \widehat{\mathrm{BP}}\cdot \mathrm{PP}$.

(2) $\widehat{\mathrm{BP}}\cdot \mathrm{C_=(PH)} = \widehat{\mathrm{BP}}\cdot \mathrm{C_=P}$ and $\widehat{\mathrm{BP}}\cdot \text{co-}\mathrm{C_=P(PH)} = \widehat{\mathrm{BP}}\cdot \text{co-}\mathrm{C_=P}$.

(3) For all k, $\widehat{\mathrm{BP}}\cdot \mathrm{MOD}_k\mathrm{P(PH)} = \widehat{\mathrm{BP}}\cdot \mathrm{MOD}_k\mathrm{P}$ and $\widehat{\mathrm{BP}}\cdot \text{co-}\mathrm{MOD}_k\mathrm{P(PH)} = \widehat{\mathrm{BP}}\cdot \text{co-}\mathrm{MOD}_k\mathrm{P}$.

DEFINITION 2.11 [5], [12]. Let $\mathbf{K}$ be any class of sets. A set L is in $\mathbf{K}$/poly if there exist a set $A \in \mathbf{K}$, an *advice function* f from natural numbers to strings, and a polynomial p such that $|f(n)| \leq p(n)$ for all n, and $L = \{ x : x\#f(|x|) \in A \}$. Let $\mathbf{K}$ be a relativizable complexity class. A set L is in almost-$\mathbf{K}$ if for almost all oracle sets X, L is in $\mathbf{K}(X)$.

Recently it has become known that for any class $\mathbf{K}$ of sets, if the class is closed downward under the polynomial-time majority reducibility, then $\mathrm{BP}\cdot\mathbf{K} \subseteq \mathbf{K}$/poly and $\mathrm{BP}\cdot \mathbf{K} \subseteq$ almost-$\mathbf{K}$; for the latter inclusion, we need some more assumptions on the class $\mathbf{K}$, e.g., it is a relativizable class and contains at most countably infinite sets. We can apply this understanding to the $\widehat{\mathrm{BP}}$-complexity classes, because in the previous works such as [17] the closure property of the class under the polynomial-time majority reducibility has been used only for amplifying the success probability in the BP-operator; on the other hand, we have already had the amplified success probability in the $\widehat{\mathrm{BP}}$-operator.

LEMMA 2.12. *For any class* $\mathbf{K}$, $\widehat{\mathrm{BP}}\cdot\mathbf{K} \subseteq \mathbf{K}$/poly.

Proof. Let L be a set in $\widehat{\mathrm{BP}}\cdot\mathbf{K}$. Then we can easily see the following: there exist a set $A \in \mathbf{K}$ and a polynomial p such that for every natural number n,

$$\Pr\{w \in \{0,1\}^{p(n)} : (\forall x, |x| = n)\, x \in L \text{ iff } x\#w \in A\} > 0.$$

From this fact, for every natural number n, we can pick up a fixed string $w_n \in \{0,1\}^{p(n)}$ such that for every string x of length n, $x \in L$ if and only if $x\#w_n \in A$. Hence, when

defining a function f by $f(n) = w_n$, we see that the function and the set A witness $L \in$ $\mathbf{K}$/poly. □

For characterizing the BP-complexity classes by means of random tally oracles, Tang and Watanabe [20] showed the following result. (Note that we mention their results by using our $\widehat{\mathrm{BP}}$-operator, though they showed it with the BP-operator.)

THEOREM 2.13 [20]. *If any given class* $\mathbf{K}$ *of sets contains at most countably many sets and is closed under the polynomial-time majority reducibility, then a set* L *is in* $\widehat{\mathrm{BP}}\cdot\mathbf{K}$ *if and only if for almost all tally sets* T, $L \in \mathbf{K} \circ \mathrm{PF}(T)$, *where* $\mathbf{K}\circ\mathrm{PF}(T)$ *is the class of sets* A *such that for some* $B \in \mathbf{K}$ *and a function* $f \in \mathrm{PF}(T)$, $A = \{ x : f(x) \in B \}$. □

In the proof of the "only if" part of this theorem, we do not need to deal only with tally sets; we may consider *any oracle sets*. (In the "if" part, we have to be concerned only with tally sets when following their proof technique; but we are not interested in the "if" part here.) Thus Tang and Watanabe could in fact prove the following lemma by their own technique (we omit the details here).

LEMMA 2.14. *Let* $\mathbf{K}$ *be any class that contains at most countably many sets. Let* L *be any set. Then, if* $L \in \widehat{\mathrm{BP}}\cdot\mathbf{K}$, *for almost all sets* X, $L \in \mathbf{K}\circ\mathrm{PF}(X)$. □

It is easy to see that for every oracle set X and every subset Q of $\mathbf{Z}$, $\mathcal{C}[Q, \mathrm{GapP}] \circ \mathrm{PF}(X) \subseteq \mathcal{C}[Q, \mathrm{GapP}(X)]$. Thus, from Lemmas 2.12 and 2.14 and Corollary 2.5, we obtain the following.

COROLLARY 2.15. *For all subsets* Q *of* $\mathbf{Z}$, $\mathcal{C}[Q, \mathrm{GapP(PH)}] \subseteq \mathcal{C}[Q, \mathrm{GapP}]/\mathrm{poly}$, *and for almost all sets* X, $\mathcal{C}[Q, \mathrm{GapP(PH)}] \subseteq \mathcal{C}[Q, \mathrm{GapP}(X)]$. *Thus, for each* $\mathbf{K}$ *of* PP, $\mathrm{C_{=}P}$, co-$\mathrm{C_{=}P}$, $\mathrm{MOD}_k\mathrm{P}$, *and* co-$\mathrm{MOD}_k\mathrm{P}$,

(1) $\mathbf{K}$(PH) $\subseteq$ $\mathbf{K}$/poly $\cap$ *almost*-$\mathbf{K}$, *and*

(2) $\mathbf{K}$(PH/poly) = $\mathbf{K}$(PH)/poly = $\mathbf{K}$/poly.

We close this section by defining a natural extension of $\mathrm{MOD}_k\mathrm{P}$ and by observing a result for the class similar to Corollary 2.5(4).

DEFINITION 2.16. Let g be a function from strings to natural numbers. We define $\mathrm{MOD}_g\mathrm{P}$ to be the class of sets L for which there exists a function $F \in \#\mathrm{P}$ such that for all x, $x \in L$ if and only if $F(x) \not\equiv 0 \pmod{g(x)}$.

COROLLARY 2.17. *For all polynomial-time computable functions* g *from strings to natural numbers such that for all* x, $g(x) \geq 2$, $\mathrm{PH} \subseteq \widehat{\mathrm{BP}}\cdot \mathrm{MOD}_g\mathrm{P}$.

Proof. For all sets $L \in$ PH, there exists a function $F \in$ GapP(PH) such that for all x, if $x \in L$, then $F(x) = 1$; otherwise, $F(x) = 0$. From this fact and Lemma 2.3, we have the corollary. □

We note that $\mathrm{MOD}_g\mathrm{P}$ cannot in general be defined as in Definition 2.2. Thus, at present we do not know whether for all g, $\mathrm{MOD}_g\mathrm{P(PH)} \subseteq \widehat{\mathrm{BP}}\cdot\mathrm{MOD}_g\mathrm{P}$, though we feel that it may be the case. In the final section, we will discuss why our proof technique cannot be applied to the class.

3. Proof of Lemma 2.3. Lemma 2.3 is obtained from the following lemma.

LEMMA 3.1. *Let* X *be an oracle set, let* F *be a function in* $\#\mathrm{P}(\mathrm{NP}(X))$, *and let* e *be a polynomial. Then there exist a function* $H \in \mathrm{GapP}(X)$ *and a polynomial* s *such that for all strings* x, $\Pr\{w \in \{0,1\}^{s(|x|)} : H(x\#w) = F(x)\} \geq 1 - 2^{-e(|x|)}$.

By applying this lemma inductively to each class in PH, we immediately obtain Lemma 2.3. The details are left to the reader. We now concentrate on proving Lemma 3.1.

Proof of Lemma 3.1. In this proof we will use some more technical lemmas. For clarity, the proofs of Lemmas 3.2–3.4 will be given after the proof of Lemma 3.1.

We first mention the following result on #P(NP(X)) that allows us to change the definition of the class.

LEMMA 3.2. *For every function* $F \in$ #P(NP(X)), *there exist a set* $A \in$ co-NP(X) *and a polynomial* p *such that for all* x, $F(x) = \|\{y \in \Sigma^{p(|x|)} : x\#y \in A\}\|$.

In the remainder of this proof, when we write $x\#y$, we assume $|y| = p(|x|)$, to simplify the argument. From the well-known characterization of co-NP(X) in [18] and [27], we have, for the set A above, a set $B \in$ P(X) and a polynomial q such that for all $x\#y$, $x\#y \in A$ if and only if there is no string $z \in \{0,1\}^{q(|x\#y|)}$ such that $x\#y\#z \in B$. By using the set B, we will later construct a function H_1 in GapP(X) and a polynomial s such that for all positive integers n,

$$\text{(A)} \qquad \Pr\{w \in \{0,1\}^{s(n)} : \text{for all } x\#y \in A \text{ of length } n, H_1(x\#y\#w) = 1, \text{ and for all } x\#y \notin A \text{ of length } n, H_1(x\#y\#w) = 0\} \geq 1 - 2^{-e(n)}.$$

By using H_1, we define the required function H as follows:

$$H(x\#w) = \sum_{y \in \{0,1\}^{q(|x|)}} H_1(x\#y\#w).$$

Then we easily see that H is in GapP(X) (provided $H_1 \in$ GapP(X)) and that H and the polynomial s above satisfy the condition of Lemma 3.1.

Now we show how to define the function H_1 and the polynomial s above. To show this, we use a consequence of Valiant and Vazirani's result [25]. Following their paper, we shall view a string $w \in \{0,1\}^m$ as a vector in GF[2]^m. We denote by $u \cdot w$ the inner product of the vectors u and w in GF[2]^m. For a string $x\#y$ of length n and a finite number of strings $w_1, \cdots, w_k \in \{0,1\}^{q(n)}$, we define a finite set $B_{x\#y}$ and $B_{x\#y}(w_1, \cdots, w_k)$ by

$$B_{x\#y} = \{z \in \{0,1\}^{q(n)} : x\#y\#z \in B\},$$

$$B_{x\#y}(w_1, \cdots, w_k) = \{z \in \{0,1\}^{q(n)} : x\#y\#z \in B \text{ and } w_1 \cdot z = \cdots = w_k \cdot z = 0\}.$$

Furthermore, we use the following notation. Let l, m be any positive integers. We denote by $\mathbf{Mat}[l,m]$ the set of all $l \times m$ matrices whose components are strings in $\{0,1\}^m$. For any matrix $W \in \mathbf{Mat}[l,m]$, we denote by $W_{j,k}$ the (j,k)-component of W. Below, we will view a matrix $W \in \mathbf{Mat}[l,m]$ as the string

$$W_{1,1}W_{1,2}\cdots W_{1,m}W_{2,1}W_{2,2}\cdots W_{2,m}\cdots W_{l,1}W_{l,2}\cdots W_{l,m},$$

which is in $\{0,1\}^{l \cdot m^2}$. Then we have the following lemma (from Valiant and Vazirani's result [25]).

LEMMA 3.3. *Let* e *be a polynomial. Then there exists a polynomial* r *such that for all strings* $x\#y$ *of length* n,

(1) *if* $x\#y \in A$, *then*

$$\Pr\{W : \|B_{x\#y}\| = 0 \text{ and } (\forall k, j, 1 \leq k \leq r(n), 1 \leq j \leq q(n)) [\|B_{x\#y}(W_{k,1}, \cdots, W_{k,j})\| = 0]\} = 1, \text{ and}$$

(2) *if $x\#y \notin A$, then*

$$\Pr\{W : \|B_{x\#y}\| = 1 \textit{ or } (\exists k, j, 1 \le k \le r(n), 1 \le j \le q(n)) [\|B_{x\#y}(W_{k,1}, \cdots, W_{k,j})\| = 1]\} \ge 1 - 2^{-e(n)},$$

where W is randomly chosen from $\mathbf{Mat}[r(n), q(n)]$ *under uniform distribution.*

Since we can take an arbitrary polynomial e in the above lemma, we obtain a stronger observation than the one above: For all positive integers n,

(B)
$$\begin{aligned}\Pr\{\ W :\ & \text{for all strings } x\#y \text{ of length } n,\\ & x\#y \in A \Rightarrow \|B_{x\#y}\| = 0 \text{ and}\\ & \quad (\forall k, j, 1 \le k \le r(n), 1 \le j \le q(n))\\ & \qquad [\|B_{x\#y}(W_{k,1}, \cdots, W_{k,j})\| = 0], \text{ and}\\ & x\#y \notin A \Rightarrow \|B_{x\#y}\| = 1 \text{ or}\\ & \quad (\exists k, j, 1 \le k \le r(n), 1 \le j \le q(n))\\ & \qquad [\|B_{x\#y}(W_{k,1}, \cdots, W_{k,j})\| = 1]\ \}\\ & \qquad \ge 1 - 2^{-e(n)},\end{aligned}$$

where W is randomly chosen from $\mathbf{Mat}[r(n), q(n)]$ under uniform distribution.

Now we define the function H_1 as follows: For all strings $x\#y$ of length n and all $W \in \mathbf{Mat}[r(n), q(n)]$,

$$H_1(x\#y\#W) = \{(G(x\#y\#\lambda) - 1) \cdot \prod_{1 \le k \le r(n), 1 \le j \le q(n)} (G(x\#y\#W_{k,1} \cdots W_{k,j}) - 1)\}^2$$

(for other strings W, we define the value of H_1 to be zero), where G is defined as follows (which gives $\|B_{x\#y}(w_1, \cdots, w_k)\|$): for all strings u,

$$G(x\#y\#u) = \begin{cases} \|B_{x\#y}\| & \text{if } |y| = r(|x|) \text{ and } u = \lambda,\\ \|B_{x\#y}(w_1, \cdots, w_k)\| & \text{if } |y| = p(|x|) \text{ and for some binary}\\ & \text{strings } w_1, \cdots, w_k \text{ with } |w_1| = \cdots = |w_k|\\ & = q(n),\ u = w_1 w_2 \cdots w_k,\\ 0 & \text{otherwise.}\end{cases}$$

Furthermore, we define the required polynomial s by $s(n) = r(n) \cdot q^2(n)$. Then we show that H_1 and s satisfy (A) mentioned previously. (In what follows, recall that we are viewing a matrix $W \in \mathbf{Mat}[r(n), q(n)]$ as a string in $\{0,1\}^{s(n)}$.) Let $x\#y$ be a string of length n and let $W \in \mathbf{Mat}[r(n), q(n)]$. If $x\#y \in A$, then $\|B_{x\#y}\| = 0$ and for all $1 \le k \le r(n)$ and all $1 \le j \le q(n)$, $\|B_{x\#y}(W_{k,1}, \cdots, W_{k,j})\| = 0$. Hence, in this case, we have $H_1(x\#y\#W) = 1$. Otherwise, $\|B_{x\#y}\| = 1$ or $\|B_{x\#y}(W_{k,1}, \cdots, W_{k,j})\| = 1$ for some k, j. Hence, in this case, we have $H_1(x\#y\#W) = 0$. From these observations, we see that the condition in (B) implies the condition in (A), and hence see that the

probability in (A) is greater than or equal to the probability in (B). Thus H_1 and s satisfy (A).

Obviously, G is in #P(X). Then, the following lemma tells us the membership of H_1 in GapP(X). The lemma draws out an essential idea developed independently by Gundermann, Nasser, and Wechsung [11] and by Ogiwara [14]. Below, we shall view a finite *multiset* of strings as a list of strings in the multiset and consider a list of strings as an element of the set $\Sigma^*(\#\Sigma^*)^*$. Hence we shall view $\Sigma^*(\#\Sigma^*)^*$ as the class of all finite multisets. We also use the ordinary set-theoretical notations for multisets.

LEMMA 3.4 [11], [14]. *Let F, G be functions in #P(X) and let $f : \Sigma^* \to \Sigma^*(\#\Sigma^*)^*$ be a polynomial-time computable function. Then, the function*

$$H(x) = \prod_{y \in f(x)} (F(y) - G(y))$$

is in GapP(X).

This completes the proof of Lemma 3.1. □

It remains to prove Lemmas 3.2–3.4. In those proofs, we will use the notations defined in the proof of Lemma 3.1.

Proof of Lemma 3.2. Let M and C be a polynomial-time bounded NOTM and an oracle set from NP(X), respectively, that witness a function F being in #P(NP(X)). Let N be a polynomial-time bounded NOTM that accepts C relative to X. We below assume, without loss of generality, that all possible computation paths of M together with possible oracle answers and those paths of N^X are encoded into binary strings in a usual manner. Let w be a computation path of M on a given input x, which includes possible oracle answers. Then we denote by $YES_M(x, w)$ (respectively, $NO_M(x, w)$) the set of query strings that are made by M along path w and whose corresponding oracle answer in w is "yes" (respectively, "no"). Now we define a set A as follows:

$$\begin{aligned} A = \{x\#w\#y_1\#\cdots\#y_m :\ & w \text{ is a computation path of } M \text{ on input } x, m \geq 0, \\ & YES_M(x, w) \text{ contains } m \text{ strings } z_1, \cdots, z_m, \\ & \text{each } y_i \text{ is the lexicographically smallest accepting} \\ & \text{computation path of } N^X \text{ on input } z_i, \text{ and} \\ & NO_M(x, w) \subseteq \overline{C}\}. \end{aligned}$$

Furthermore, by a standard padding argument, we can so easily adjust the definition of A (without changing its complexity) that for some polynomial p and any strings x and u, if $x\#u \in A$, then $|u| = p(|x|)$. Then, we see that A is in co-NP(X) and for every string x,

$$F(x) = \|\{u \in \Sigma^{p(|x|)} : x\#u \in A\}\|. \qquad \square$$

Proof of Lemma 3.3. Let A and B be the same sets as in Lemma 3.1. In their paper [25] Valiant and Vazirani showed the following claim. (Note that in the claim, we modify their original result by using our present notations.)

CLAIM 1 [25]. *For all strings $x\#y$ of length n,*

(1) *if $x\#y \in A$, then*

$$\Pr\{w_1, \cdots, w_{q(n)} : (\forall k, 0 \leq k \leq q(n))[\|B_{x\#y}(w_1, \cdots, w_k)\| = 0]\} = 1, \textit{ and}$$

(2) *if $x\#y \notin A$, then*

$$\Pr\{w_1, \cdots, w_{q(n)} : (\exists k, 0 \leq k \leq q(n))[\|B_{x\#y}(w_1, \cdots, w_k)\| = 1]\} \geq \tfrac{1}{4},$$

where each w_i is randomly chosen from $\{0,1\}^{q(n)}$ under uniform distribution. (Recall that q is the polynomial mentioned in Lemma 3.1.)

Repeating the random process in Claim 1 above and taking the disjunction of the outcomes, we can amplify the probability in (2) without changing the probability in (1). To state this more precisely, if we repeat the random process $r(n)$ times where r is an arbitrary polynomial, then we have the following:

(1′) if $x\#y \in A$, then

$$\Pr\{W : \|B_{x\#y}\| = 0 \text{ and } (\forall k, j, 1 \le k \le r(n), 1 \le j \le q(n))[\|B_{x\#y}(W_{k,1}, \cdots, W_{k,j})\| = 0]\} = 1,$$

and

(2′) if $x\#y \notin A$, then

$$\Pr\{W : \|B_{x\#y}\| = 1 \text{ or } (\exists k, j, 1 \le k \le r(n), 1 \le j \le q(n))[\|B_{x\#y}(W_{k,1}, \cdots, W_{k,j})\| = 1]\} \ge 1 - (3/4)^{r(n)},$$

where W is randomly chosen from $\mathbf{Mat}[r(n), q(n)]$ under uniform distribution. Thus, when we take the polynomial r such that $(\frac{3}{4})^{r(n)} \le 2^{-e(n)}$ for all $n > 0$, we have this lemma. □

Proof of Lemma 3.4. Taking the expansion of $H(x)$, we can express $H(x)$ by

$$H(x) = \sum_{S \subseteq f(x)} \left(\prod_{y \in S} F(y) \right) \cdot (-1)^{\|f(x)-S\|} \cdot \left(\prod_{y \in f(x)-S} G(y) \right),$$

where we define $\prod_{y\in\emptyset} F(y) = \prod_{y\in\emptyset} G(y) = 1$. For each additive term of the above expression, if $\|f(x) - S\|$ is odd, then the term is negative; otherwise, it is positive. So, by separating both cases from each other, we can express $H(x)$ by

$$H(x) = \sum_{S\subseteq f(x), \|f(x)-S\|:\text{even}} (\prod_{y\in S} F(y))(\prod_{y\in f(x)-S} G(y)) - \sum_{S\subseteq f(x), \|f(x)-S\|:\text{odd}} (\prod_{y\in S} F(y))(\prod_{y\in f(x)-S} G(y)).$$

It is easy to see that each summation in the last expression is realized as a #P(X) function. Thus H is in GapP(X). □

4. Concluding remarks. We showed that all the known counting classes are at least as hard as the polynomial-time hierarchy; that is, all sets in the polynomial-time hierarchy are randomly reducible to sets in the counting classes with exponentially small error probability. A crucial point in showing these results is that every counting class of concern in this paper can be characterized in terms of a simple predicate over #P functions (alternatively, GapP functions). In fact, we showed, as a more general result, that the same relationship as above holds for all counting classes that can be defined as in Definition 2.2 by using GapP functions.

Nonetheless we can obtain a broader family of counting classes, as Fenner, Fortnow, and Kurtz have defined the gap-definable classes in [8]. As an immediate question related to this work, it may be asked whether our main result remains true for all of the

gap-definable classes. Unfortunately we have not been able to settle the question. But, by intuition, it seems unlikely to be the case. As an example more concrete than our intuition, we here consider SPP [8], that is, the class of sets L for which there exists a function F in GapP such that for every x, $x \in L$ if and only if $F(x) = 1$ and $x \in L$ if and only if $F(x) = 0$. In the definition of SPP, there exists a promise, as in the case of UP [23] and Few [7], that the GapP function never takes the value other than zero or 1. Because of the existence of such a promise, we conjecture that SPP is not as computationally hard as the polynomial-time hierarchy (at least in the sense of the present work). In [8] SPP has been shown to be a gap-definable class. Thus we think that our main result cannot be extended to the gap-definable classes.

As mentioned at the end of §2, $\mathrm{MOD}_g\mathrm{P}$ is out of our family and is a typical example to which our proof technique could not be applied. In order to establish the relationship $\mathrm{MOD}_g\mathrm{P(PH)} \subseteq \widehat{\mathrm{BP}}\cdot\mathrm{MOD}_g\mathrm{P}$, we must probably show, for all L in $\mathrm{MOD}_g\mathrm{P(PH)}$, the existence of H_1 in GapP and a polynomial s such that for all x, $x \in L$ if and only if for (intuitively) almost all w of $\{0,1\}^{s(|x|)}$, $H_1(x\#w) \not\equiv 0 \pmod{g(x\#w)}$. By using our argument, we can find a function $H_2 \in$ GapP such that for all x, $x \in L$ if and only if for almost all w of $\{0,1\}^{s(|x|)}$, $H_2(x\#w) \not\equiv 0 \pmod{g(x)}$, but in general, we cannot know how $g(x)$ is related to $g(x\#w)$ nor how to construct the function H_1 from H_2. This is why we could not establish the relationship, although we conjecture it is the case.

We think it is still important to find much closer relationships between counting classes and classes in PH. As mentioned in §1, it seems very hard to show a new inclusion relationship. Thus we mainly concentrate on some reducibility questions. In particular, it is more important to know whether all sets in PH (or a class in PH) are polynomial-time reducible to sets in $\mathrm{C_=P}$ or $\mathrm{MOD}_k\mathrm{P}$ under *deterministic reducibilities* such as Turing or truth-table ones. Considering the current status on this question, we think it still nice to show that all sets in Σ_2^{P} are polynomial-time Turing reducible to sets in $\mathrm{C_=P}$ (or in $\mathrm{MOD}_k\mathrm{P}$), if it holds, or to find oracles separating those classes.

Acknowledgment. This work was done while the authors were visiting the Department of Mathematics at the University of California at Santa Barbara. We are very grateful to Professor Ronald V. Book for his advice and hospitality during that time. Many thanks to Richard Beigel, Lance Fortnow, and Jun Tarui for their valuable suggestions and discussions, and to the anonymous referees for their suggestions for improving our presentation.

REFERENCES

[1] R. BEIGEL, *Relativizing counting classes: Relations among thresholds, parity, and mods*, J. Comput. System Sci., 42 (1990), pp. 76–96.

[2] ______, *Polynomial interpolation, threshold circuits, and the polynomial-time hierarchy*, preliminary version, 1990 unpublished manuscript.

[3] R. BEIGEL, L. A. HEMACHANDRA, AND G. WECHSUNG, *On the power of probabilistic polynomial-time*: $\mathrm{P}^{NP[log]} \subseteq \mathrm{PP}$, Proc. 4th IEEE Conference on Structure in Complexity Theory, 1989, pp. 225–227.

[4] R. BEIGEL, J. GILL, AND U. HERTRAMPF, *Counting classes: Thresholds, parity, mods, and fewness*, in Proc. 7th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 415, 1990, Springer-Verlag, New York, pp. 49–57.

[5] C. H. BENNETT AND J. GILL, *Relative to a random oracle*, $\mathrm{P}^A \neq \mathrm{NP}^A \neq \text{co-NP}^A$ *with probability* 1, SIAM J. Comput., 10 (1981), pp. 96–113.

[6] R. BEIGEL, N. REINGOLD, AND D. SPIELMAN, PP *is closed under intersection*, Tech. Report YALE/DCS/TR-803, Department of Computer Science, Yale University, June 1990; also Proc. 23rd ACM Symposium on Theory of Computing, 1991, pp. 1–9.

[7] J. CAI AND L. A. HEMACHANDRA, *On the power of parity polynomial time*, Proc. 6th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci. 349, 1989, Springer-Verlag, New York, pp. 229–240; also Math. Systems Theory, 23 (1990), pp. 95–106.
[8] S. A. FENNER, L. J. FORTNOW, AND S. A. KURTZ, *Gap-definable counting classes*, Proc. 6th IEEE Conference on Structure in Complexity Theory, 1991, pp. 30–42.
[9] L. FORTNOW AND N. REINGOLD, PP *is closed under truth-table reductions*, Tech. Report 90-30, Department of Computer Science, University of Chicago, Chicago, IL, September 1990; also Proc. 6th IEEE Conference on Structure in Complexity Theory, 1991, pp. 13–15.
[10] J. GILL, *Computational complexity of probabilistic Turing machines*, SIAM J. Comput., 6 (1977), pp. 675–695.
[11] T. GUNDERMANN, N. A. NASSER, AND G. WECHSUNG, *A Survey on Counting Classes*, Proc. 5th IEEE Conference on Structure in Complexity Theory, 1990, pp. 140–153.
[12] R. KARP AND R. LIPTON, *Some connections between nonuniform and uniform complexity classes*, Proc. 12th ACM Symposium on Theory of Computing, 1980, pp. 302–309.
[13] J. KÖBLER, U. SCHÖNING, S. TODA, AND J. TORAN, *Turing machines with few accepting computations and low sets for* PP, Proc. 4th IEEE Conference on Structure in Complexity Theory, 1989, pp. 208–215.
[14] M. OGIWARA, *On the computational power of exact counting*, unpublished manuscript, 1990.
[15] C. H. PAPADIMITRIOU AND S. ZACHOS, *Two remarks on the power of counting*, Proc. 6th GI Conference on Theoretical Computer Science, Lecture Notes in Comput. Sci. 145, Springer-Verlag, New York, 1983, pp. 296–276.
[16] J. SIMON, *On the difference between one and many*, in Proc. 4th Colloquium on Automata, Languages and Programming, Lecture Notes in Comput. Sci. 52, Springer-Verlag, New York, 1977, pp. 480–491.
[17] U. SCHÖNING, *Probabilistic complexity classes and lowness*, Proc. 2nd IEEE Conference on Structure in Complexity Theory, 1987, pp. 2–8; also J. Comput. System Sci. 39 (1988), pp. 84–100.
[18] L. J. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1–22.
[19] J. TARUI, *Randomized polynomials, threshold circuits, and the polynomial hierarchy*, Proc. 8th Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Comput. Sci., 480, 1991, pp. 238–250.
[20] S. TANG AND O. WATANABE, *On tally relativizations of* BP*-complexity classes*, SIAM J. Comput., 18 (1988), pp. 449–462.
[21] S. TODA, *On the computational power of* PP *and* $\oplus$P, Proc. 30th IEEE Symposium on Foundations of Computer Science, 1989, pp. 514–519.
[22] J. TORAN, *Structural properties of the counting hierarchies*, Ph.D. thesis, Departament de Llenguatges i sistemes informàtics, Universtat Polytècnica de Catalunya, Barcelona, Spain, 1988.
[23] L. G. VALIANT, *Relative complexity of checking and evaluating*, Inform. Process. Lett., 5 (1976), pp. 20–23.
[24] ______, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189–201.
[25] L. G. VALIANT AND V. V. VAZIRANI, NP *is as easy as detecting unique solutions*, Theoret. Comput. Sci., 47 (1986), pp. 85–93.
[26] K. WAGNER, *The complexity of combinatorial problems with succinct input representation*, Acta Inform., 23 (1986), pp. 325–356.
[27] C. WRATHALL, *Complete sets and the polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 23–33.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 329–338, April 1992

© 1992 Society for Industrial and Applied Mathematics
009

LOWER BOUNDS FOR THRESHOLD AND SYMMETRIC FUNCTIONS IN PARALLEL COMPUTATION*

YOSSI AZAR†

Abstract. The family of decision problems of the threshold languages L_g is considered. A threshold language L_g is the set of n bit vectors having at least $g(n)$ "1"s. Using a new technique for controlling the size and structure of a hypergraph by a potential function, lower bounds are proven for these decision problems on a PRIORITY PRAM with m shared memory cells and any polynomial number of processors. The lower bounds are almost tight for the admissible range ($m \leq n^{\epsilon}$). By combining these results with the results of Vishkin and Wigderson and the results of Li and Yesha, this paper is able to show a complexity gap between an m cell PRIORITY PRAM having an exponential (or unlimited) number of processors and one having only a polynomial number. A consequence of these results is that PRIORITY PRAM and ARBITRARY PRAM with m shared memory cells and any given polynomial number of processors have the same power (up to a small factor) for computing symmetric functions.

Key words. threshold, PRAM, lower bounds, symmetric functions, parallel computation

AMS(MOS) subject classifications. 68Q10, 68R10, 05C65

1. Introduction. This paper considers the PRAM model with limited shared memory. A PRAM consists of processors $P(i)$, $i = 1, \cdots, p$, and shared memory cells $C(i)$, $i = 1, \cdots, m$, through which the processors communicate. Since we assume that the number of shared memory cells (m) is smaller than the input size (n), there are also n read-only input cells (ROM) $X(1), \cdots, X(n)$. There is a hierarchy among the different PRAM models in accordance with the way they resolve write conflicts. The strongest model is the PRIORITY model, where the minimal index processor (among those attempting to write to the same cell) succeeds. We prove lower bounds for the PRIORITY model with m shared memory cells, denoted by PRIORITY(m). In order to get more powerful lower bounds, we assume that each cell in the shared memory can accommodate strings of arbitrary length.

For a given function $g = g(n)$, the threshold language L_g (see [LY2]) is defined by

$$L_g = \left\{ (x_1, \cdots, x_n) | x_i \in \{0,1\} \ (1 \leq i \leq n) \ \text{and} \ \sum_{i=1}^{n} x_i \geq g(n) \right\}.$$

By symmetry we can always assume that $g(n) \leq n/2$. The language L_g, where $g(n) = n/2$, is called the MAJORITY language. The complexity of the family of threshold languages has immediate implications on the complexity of symmetric functions.

1.1. Background and previous work. The first to consider this model were Vishkin and Wigderson [VW]. They proved an $\Omega(\sqrt{n/m})$ lower bound for computing MAJORITY ($g = n/2$) on PRIORITY(m). Their lower bound does not depend on the number of processors and is tight for all $m \leq n/\log^2 n$. The upper bound can be easily obtained using $\sqrt{nm}$ processors. For general $g(n)$ their method produces an $\Omega(\sqrt{g(n)/m})$ lower bound. They argue that the case of a single shared memory cell ($m = 1$), or of a constant number, is not only interesting from the theoretical point of view, but is also well founded in practice. For example, the "Ethernet" can be considered as a PRAM with a

*Received by the editors April 12, 1990; accepted for publication (in revised form) May 21, 1991. This work was supported by a Weizmann Fellowship and by Office of Naval Research contract N00014-88-K-0166.

†Computer Science Department, Stanford University, Stanford, California 94305-2140. Present address, DEC-Systems Research Center, 130 Lytton Ave., Palo Alto, California 94301.

single shared memory cell. Vishkin and Wigderson also point out that in [GGKMRS], [K], and [V] it is implied that the size of the shared memory may determine the hardware feasibility of the parallel machine. More lower bounds for PRAM with small shared memory appear in [B], [FMW], [FRW], and others.

Li and Yesha made more progress on analyzing the complexity of threshold languages in this model. In [LY2] they considered the threshold languages L_g for $g = o(n)$. They noticed that the $\Omega(\sqrt{g})$ lower bound ($m = 1$) can be matched using $\binom{n}{g^{0.5}}$ processors. Since this number is exponential in n, they suggested that it is of interest to find more accurate bounds for smaller number of processors. They present a better lower bound only for the special case where the number of processors is *linear* and $g = o(n^{1/4})$. Specifically, they showed that for $g = o(n^{1/4})$, a PRIORITY(m) with $O(n)$ processors requires depth $\Omega(g/m)$ to recognize L_g. For $m = 1$, a matching upper bound can be easily obtained.

However, the general problem remained open, i.e., to find the complexity of the family of the threshold languages for any polynomial number of processors. Moreover, the lower bound in [LY2] holds only for the subfamily where $g = o(n^{1/4})$. Thus, it is desirable to determine the complexity over the entire range of g. From the fact that the threshold can be found for any g in $O(\sqrt{n})$ steps, even for $m = 1$ and using only $\sqrt{n}$ processors (see [VW]), it follows that the lower bound of $\Omega(g)$ cannot be obtained for every $g = o(n)$. Hence, it is of interest to know for which range the $\Omega(g)$ lower bound holds and how it changes for bigger g.

1.2. Our results. We address all these problems and prove a lower bound for the general case. We assume without loss of generality that $g \leq n/2$, since the complexity of g and $n-g$ is the same because of symmetry. We prove that $\Omega(g/m)$ rounds are necessary for solving the L_g decision problem for any polynomial number of processors and for the admissible range of g. More precisely, we show that for any $\epsilon > 0$, and for any constant $c > 0$, a PRIORITY(m) with $O(n^c)$ processors requires depth $\Omega(\min(\frac{g}{m}, \frac{n^{1/2-\epsilon}}{m}))$. This bound is tight for $m = 1$ and $g \leq n^{1/2-\epsilon}$, since it can easily be obtained by using n/g processors. It is almost tight (up to a factor of $n^{2\epsilon}$) for the remaining range of g and for all $m \leq n^{\epsilon}$. This follows from the existence of a simple algorithm that runs using one shared memory cell in $O(n^{1/2})$ rounds and $n^{1/2}$ processors, and due to the fact that the lower bound is reduced by a factor of at most m for m cell PRAMs, compared to a single shared memory cell.

Our results thus determine the complexity of the family of the threshold problems when a polynomial number of processors is available. Moreover, we conclude that the running time of an algorithm that uses a polynomial number of processors will not be better than the running time of one that uses only a linear number of processors. Our results also show that the complexity of the problem with a polynomial number of processors is different from the complexity in the exponential case. Thus, there is a quadratic gap between the cases. For example, for $m = 1$ and exponential number of processors the complexity is $\Theta(\sqrt{g})$; however, for the polynomial range, we prove that it behaves in the following curious way. It is g for $g \leq n^{1/2}$, and it becomes $n^{1/2}$ for the remaining domain up to a factor of less than n^{ϵ}.

By combining our results with those of [LY2], we show that for any polynomial number of processors and $m \leq n^{\epsilon}$, PRIORITY(m) and ARBITRARY(m) require the same time complexity (up to a factor of at most $n^{2\epsilon}$) for computing any symmetric function. For $m = 1$ we prove that the two models are equivalent for a large family of symmetric functions. These results are based on the strong connection between computing sym-

metric functions and threshold decision problems (see [LY2]).

Let us briefly elaborate on the techniques that are used to prove our main lower bound. The main new tool is the use of a potential function on a dynamic hypergraph that changes at every step according to the actions of the processors. The set of vertices of the hypergraph corresponds to the ROM cells. The hyperedges reflect, at each step, the main set of constraints defining the legal inputs at that step. We perform several types of transformations on the hypergraph so as to maintain the following invariant: all the legal inputs have the same history. The construction also guarantees that the set of legal inputs contains at least one input in L_g and at least one input that is not in L_g. Interestingly, the structure and size of the hypergraph are controlled by the potential function. These structures also serve to control the information that each processor can deduce from the ROM and the shared memory.

2. The main lower bound. Each step of the PRAM computation consists of four phases: each processor $P(i)$ reads from some shared memory cell $C(j)$, reads from some ROM cell $X(k)$, performs a computation, and may try to write into some shared memory cell $C(l)$. The actions and the next state of each processor depend on the current state, the values read from the ROM, and the shared memory. See [VW] and [LY] for formal definitions.

The strongest model, the PRIORITY model, is considered. Thus, the minimal index processor succeeds in writing when write conflicts occur. Recall that PRIORITY(m) denotes the model with m shared memory cells. Since each round of PRIORITY(m) can be simulated by $2m$ rounds of PRIORITY(1), all the lower bounds of $m = 1$ hold for PRIORITY(m) when divided by $2m$. Thus, we can concentrate in the case where $m = 1$.

We state our main theorem.

MAIN THEOREM. *Let k be any positive integer.* PRIORITY(1) *requires $\Omega(g/k)$ rounds to recognize L_g using p processors when the following inequality holds:*

$$p[2^{k+4}g]^{2k+1} \leq n^{k+1}. \tag{2.1}$$

COROLLARY 1. *Let $\epsilon > 0$ and $d = o(\log n)$. Then $\Omega(g/d)$ rounds are needed in order to recognize L_g using $p = O(n^d)$ processors for $g \leq n^{1/2-\epsilon}$.*

Proof. For $g \leq n^{1/2-\epsilon}$ we can choose $k = Cd$, where C is large enough constant such that (2.1) holds for large enough n. □

An important special case of Corollary 1 is when d is constant.

COROLLARY 2. *For any polynomial number of processors $\Omega(g)$ rounds are needed to recognize L_g for $g \leq n^{1/2-\epsilon}$.*

A matching upper bound can be easily obtained using a linear number of processors as follows. Associate a processor with each ROM cell. At each step each processor whose input bit is 1 tries to write its identity into the shared memory cell. Then it reads from this cell and checks if it succeeded in writing. If it did succeed, then it halts. An input is in L_g if and only if the algorithm is alive for at least g steps.

Thus, we cannot improve the asymptotic running time using any polynomial number of processors compared to the case of using only a linear number of them. The results also separate, as we already mentioned, between the power of a polynomial number of processors and the power of an exponential number, since the complexity in the latter case is $\Theta(\sqrt{g})$.

Note that the complexity is monotonic (up to a constant factor) in g for $g \leq n/2$, since we can pad the input vector by 1's. We conclude a lower bound for the remaining range of g.

COROLLARY 3. *For any polynomial number of processors $\Omega(n^{1/2-\epsilon})$ rounds are needed to recognize L_g for $g \geq n^{1/2-\epsilon}$.*

This bound is almost tight as it is easy to find the exact threshold (to sum up the bits) in $O(n^{1/2})$ rounds using $n^{1/2}$ processors as follows. Partition the input into $n^{1/2}$ blocks, each of size $n^{1/2}$, and associate a processor with each block. In the first $n^{1/2}$ rounds each processor sums up the bits in its block. In the next $n^{1/2}$ rounds each processor in turn adds its result to the shared memory cell. This results in the bit sum.

In fact, we can further decrease the n^ϵ gap between the upper and lower bound for $g \geq n^{1/2-\epsilon}$. By choosing $k = \Theta(\sqrt{\log n})$ we can obtain from our main theorem an $\Omega(g/k)$ lower bound for $g = n^{1/2}/2^{\Theta(\sqrt{\log n})}$. This yields a gap of $O(\sqrt{\log n})$ for $n^{1/2-\epsilon} \leq g \leq n^{1/2}/2^{\Theta(\sqrt{\log n})}$ and for $g > n^{1/2}/2^{\Theta(\sqrt{\log n})}$ the gap is reduced to at most $2^{\Theta(\sqrt{\log n})}$.

Corollary 1 combined with the result of [VW] yields the following lower bound.

COROLLARY 4. *For $g \leq n^{1/2-\epsilon}$ and $d = o(\log n)$, $\Omega(g/d + \sqrt{g})$ rounds are needed to recognize L_g using $O(n^d)$ processors.*

This result matches up to a constant factor the [LY2] upper bound of $O(g/d + \sqrt{g})$ using $\binom{n}{d} = O(n^d)$ processors.

Recall that PRIORITY(1) can simulate each step of PRIORITY(m) in at most $2m$ steps. This yields the following corollary.

COROLLARY 5. *All of the above lower bounds divided by $2m$ apply to* PRIORITY*(m). Thus, for $m \leq n^\epsilon$ the bounds are tight up to a factor of at most $n^{2\epsilon}$ (in fact, up to at most n^δ for any $\delta > \epsilon$).*

3. The proof of the Main Theorem. In this section we prove the main theorem. In the first part the main inductive hypothesis is presented and proved. In the second part the proof of the main theorem is completed using the main inductive hypothesis and some of the transformations defined in the first part.

First let us assume that $p \geq n^{1/2}$, as for the easy case $p \leq n^{1/2}$ the complexity for every g is clearly $\Theta(n/p)$, which is more than what the theorem claims in this case. Let k be an arbitrary positive integer.

We start with some definitions. Define the history of a computation through t steps as a vector $H_1, \cdots, H_t$, where H_i is the contents of the shared memory at step i. For $t = 0, 1, \cdots$ the adversary defines collection of inputs I_t on which M has the same history through step t ($I_t \subseteq I_{t-1}$ for $t \geq 1$). Whenever $t < g/k$, then I_t contains one input in L_g and one not in L_g. Thus the algorithm cannot recognize L_g in depth t.

Let $I_0 = \{0,1\}^n$. We will define sets $B_t \subseteq \{1, \cdots, n\}$, $B_{t-1} \subseteq B_t$. The hypergraph is defined on the vertices $\{1, \cdots, n\}$. At stage t the edges of the hypergraph will be $F_t = \cup_{i=1}^{k+1} F_t^i$, where F_t^i is a set of subsets (hyperedges), each of size i, of the vertices. Let $B_0 = \phi$ and $F_0 = \phi$.

Intuitively B_t is the set of indices of cells whose input bits are fixed to be 1 at the end of step t. A hyperedge (sometimes referred to as an edge) in the hypergraph corresponds to a constraint on the input. The cells, whose indices are the vertices of the hyperedge, cannot all contain 1. In particular, F_t^1 is the set of indices (edges of size 1) of the cells whose bits are fixed to be zero. Furthermore, we do not let any processor see more than k 1's from the input cells (in addition to the 1's in the set B_t, which are themselves known to all the processors). This is done by constructing the hypergraph edges (constraints). Let

$$q = \frac{n}{g2^{k+4}} .$$

Define a potential function W on the hypergraph F,

$$W(F) = \sum_{i=1}^{k+1} q^{-(i-1)} |F^i| .$$

3.1. The main inductive hypothesis. *The main inductive hypothesis* for the end of step $t-1$ is the following:

(I1) All the inputs in I_{t-1} have the same history through step $t-1$.

(I2) On all the inputs in I_{t-1} and for any processor $P(j)$ the set $\{i | i \notin B_{t-1}$ and $P(j)$ read a value 1 from $X(i)$ during steps 1 to $t-1\}$ has a cardinality of at most k.

(I3) $|B_{t-1}| \leq k(t-1)$.

(I4) $W(F_{t-1}) \leq q^{-k} \frac{p}{k!}(t-1)^{k+1} + 2^k q(t-1) + (t-1)^2$.

(I5) For any $s < r \leq k+1$ and all j_i, $1 \leq i \leq r$, if $[\{j_1, \cdots, j_r\} \in F_{t-1}^r$ and for $1 \leq i \leq s$ $j_i \in B_{t-1}]$ then $\{j_{s+1}, \cdots, j_r\} \in F_{t-1}^{r-s}$.

(I6) $I_{t-1} = \{(x_1, \cdots, x_n) \in I_0 | \, x_j = 1$ for all j in B_{t-1}, and $x_{j_1} x_{j_2} \cdots x_{j_i} = 0$ for any $1 \leq i \leq k+1$ and all $\{j_1, \cdots, j_i\} \in F_{t-1}^i\}$.

Clearly the main inductive hypothesis holds for $t-1=0$. We have to prove it for t assuming it is true for $t-1$. We start step t by a *constraint-adding* stage. The goal of this stage is to satisfy (I2) for the end of step t. For that we add to F_{t-1}^{k+1} a set A_t of hyperedges, each of size $k+1$. Let $A_t = \{\{j_1, \cdots, j_{k+1}\} |$ for $i = 1, \cdots, k+1$, j_i is not in B_{t-1} and on some input in I_{t-1} some processor read 1 from all the $X(j_i)$ during steps 1 through $t\}$.

We would like to estimate the size of A_t in order to evaluate the change in W. Let $R_t(l)$ be all the sets $\{j_1, \cdots, j_{k+1}\}$ such that the j_i's are not in B_{t-1}, and on some input in I_{t-1} processor $P(l)$ read a value 1 from $X(j_i)$ for all $1 \leq i \leq k+1$ during steps 1 to t.

For $1 \leq t_1 < t_2 < \cdots < t_k \leq t-1$ let $I_{t-1}^l(t_1, \cdots, t_k)$ be all the inputs $x \in I_{t-1}$ such that on input x processor $P(l)$ read at step t_i, for all $1 \leq i \leq k$, a value 1 from some ROM location $X(j_i)$, j_i is not in B_{t-1} and all the j_i's are different.

CLAIM 1. *All the inputs in $I_{t-1}^l(t_1, \cdots, t_k)$ contribute at most one set to $R_t(l)$.*

Proof. The basic idea of the proof is to prove by induction on j, $j \leq t$, that during steps 1 to j on all the inputs in $I_{t-1}^l(t_1, \cdots, t_k)$, the sequence of ROM locations read by $P(l)$ is the same, and unless $j = t$ their values are also the same. The claim follows easily from this assertion and from (I2).

The assertion is clearly true for $j = 0$. Assume that the assertion holds up to step $j-1$. Thus, all the inputs in $I_{t-1}^l(t_1, \cdots, t_k)$ have the same history and on all these inputs the same sequence of values was read by $P(l)$ from the ROM. Thus, at step j on these inputs, $P(l)$ will read from the same location (denoted by j^*) from the ROM. It is left to show that $P(l)$ will also read the same value when $j < t$.

We consider four cases. If $j = t_i$ for some $i = 1, \cdots, k$, then by definition the value is always 1. If $j^* \in B_{t-1}$, then by (I6) the value is also necessarily 1. If $j^* = t_i^*$ for some $i = 1, \cdots, k$, where t_i^* is the common (by induction) location in the ROM from which $P(l)$ read at step t_i on all the inputs $I_{t-1}^l(t_1, \cdots, t_k)$, then clearly the value is 1 for all these inputs. Otherwise, the value has to be zero: a value 1 would contradict (I2), since $t_1^*, \cdots, t_k^*, j^*$ would be a sequence of $k+1$ different places not in B_{t-1} from which $P(l)$ read a value 1. □

Since $A_t \subseteq \cup_{l=1}^p R_t(l)$, Claim 1 implies that

$$|A_t| \leq p \binom{t-1}{k} \leq \frac{p}{k!}(t-1)^k .$$

Let G'_t be the hypergraph after this stage. Hence

$$W(G'_t) \leq W(F_{t-1}) + q^{-k}\frac{p}{k!}(t-1)^k .$$

We need to remark that at almost any point in the proof the hypergraph might have redundant edges, i.e., edges that correspond to redundant constraints. Specifically, if for some hyperedges X and Y, $X \subset Y$, then we may omit Y. This neither changes the legal inputs nor increases the potential function. However, assumption (I5) should be interpreted in the following way. The subset $\{j_{s+1}, \cdots, j_r\}$ (in (I5)) is not required to be an edge in the hypergraph, but rather some subset of it is required to be an edge.

We define on the hypergraph F the *weighted sunflower transformation* as follows. For a current hypergraph denote by $d^r_s\{j_1, \cdots, j_s\}$ the number of hyperedges of size r that contain $\{j_1, \cdots, j_s\}$ as a subset ($s < r$). If for some s, some set $\{j_1, \cdots, j_s\}$ which is not an edge in the current F satisfies

$$\sum_{r=s+1}^{k+1} q^{-(r-1)} d^r_s\{j_1, \cdots, j_s\} > q^{-(s-1)},$$

then we add a new edge $\{j_1, \cdots, j_s\}$ to the current F and omit all the edges that contain this set as a subset. In this way we create a current F. We repeat the above transformation on the current hypergraph as much as possible in any arbitrary order. Clearly, the number of hyperedges decreases at each transformation and therefore the process is final. The potential function W on F does not increase during this process, since the potential of a new edge is at most the sum of the potentials of the edges that it has replaced. Let G_t denote the hypergraph at the end of this process. Thus,

$$W(G_t) \leq W(G'_t) \leq W(F_{t-1}) + q^{-k}\frac{p}{k!}(t-1)^k.$$

Let $I'_t = \{(x_1, \cdots, x_n) \in I_0 |\ x_j = 1$ for all j in B_{t-1}, and $x_{j_1} x_{j_2} \cdots x_{j_i} = 0$ for any $1 \leq i \leq k$ and all $\{j_1, \cdots, j_i\} \in G^i_t\}$.

Clearly $I'_t \subseteq I_{t-1}$, since the constraint-adding stage as well as the weighted sunflower transformations could just restrict the legal inputs. Now there are two possible cases. The first is that no processor writes at step t on any input in I'_t. In this easy case we let $B_t = B_{t-1}$. The general case is when there exists a processor that writes at step t on some input in I'_t. Consider the set of processors which write at step t for some input in I'_t. Let $P(l)$ be the minimum index processor in this set. Suppose that $P(l)$ writes at step t on $x \in I'_t$. Let U_t, V_t, respectively, be the set of ROM locations from which $P(l)$ has read a value 0, 1, respectively, on input x by step t. Clearly $|U_t| \leq t$. In order to force $P(l)$ to write on all legal inputs (and by that (I1) will hold at the end of step t) we need to fix the bits of V_t to be 1 and of U_t to be zero. We let $B_t = B_{t-1} \cup V_t$; later in the proof (after the spreading transformation) we will force the bits of U_t to be zero by adding each bit as an edge to the hypergraph. By the definition of B_t and I'_t and by (I2), (I5) (both for $t-1$), we can easily conclude that

$$|B_t - B_{t-1}| \leq k,$$

and thus (I3) will hold at the end of step t.

Now the edges (constraints) in G^i_t should be changed in accordance with the new information in order to satisfy (I5). This is called the *spreading transformation*. For any s let $Z = \{j_1, \cdots, j_s\}$ be a set of any s elements in $B_t - B_{t-1}$. If it is a subset of any edge

in G_t^r $(r > s)$, then we must create a new edge of the $r - s$ remaining elements, and add it to the hypergraph. Moreover, since

$$\sum_{r=s+1}^{k+1} q^{-(r-1)} d_s^r\{j_1, \cdots, j_s\} \leq q^{-(s-1)}$$

(otherwise, it would contradict the weighted sunflower transformations), then

$$\sum_{r=s+1}^{k+1} q^{-((r-s)-1)} d_s^r\{j_1, \cdots, j_s\} \leq q^{-(s-1)} q^s = q \, .$$

Hence, the total potential of the edges that were created by the set Z is at most q. We perform the spreading transformation simultaneously for all the subsets of $B_t - B_{t-1}$ (at most 2^k subsets) and conclude that the potential function is increased by at most $2^k q$. Define F_t to be the hypergraph after the spreading transformation union with $|U_t| \leq t$ sets, each of size 1, of all the individual elements of U_t. Clearly F_t satisfies (I5). Moreover, each of these one-element sets adds at most 1 to the potential function and thus

$$W(F_t) \leq W(G_{t-1}) + 2^k q + t \, .$$

Therefore

$$W(F_t) \leq W(F_{t-1}) + q^{-k} \frac{p}{k!} (t-1)^k + 2^k q + t \, ,$$

which yields (I4) for the end of step t. Finally, we define I_t according to (I6) with t instead of $t - 1$.

This completes step t. We can easily verify that the main inductive hypothesis holds for t, since each assumption was satisfied at some point in the proof and remained satisfied henceforth.

3.2. Proving the lower bound using the main inductive hypothesis. We will prove that the algorithm cannot stop in t steps for $t < g/k$. Fix some $T < g/k$. We first look at the following input: $x_i' = 1$ for $i \in B_T$ and $x_i' = 0$ otherwise. Clearly this input belongs to I_T. By (I3), $|B_T| < g$ and therefore it is not in L_g. Constructing an input in I_T which is also in L_g is more complicated. We start with $x_i'' = 1$ for $i \in B_T$, $x_i'' = 0$ for $i \in F_T^1$. We need to find $g - |B_T| \leq g$ other locations with value 1 that will be consistent with all the constraints of F_T. More precisely, we have to find an *independent set* of size g in the hypergraph, which is a set of vertices of size g such that each edge of the hypergraph contains at least one vertex not in this set. If such an independent set exists, we will set the input bits that correspond to those vertices to be 1 (in addition to the input bits of the set B_t), and the remaining vertices to zero. This will define an input in I_T which is in L_g and will complete the proof.

First we can easily check that inequality (2.1) yields (as $p \geq n^{1/2}$) that

$$g \leq n^{1/2}/32$$

and by definition of q $(q = \frac{n}{g2^{k+4}})$ and inequality (2.1),

$$q^{-k} \frac{p}{k!} g^{k+1} \leq n/16 \, .$$

Hence, easy computation shows that for $t \leq g$,

$$W(F_t) \leq n/4 .$$

Constructing an independent set consists of g steps. These steps are independent of the actions of the processors after step T. However, for simplicity of notation these pseudosteps are referred to as steps $t = T+1, \cdots, T+g$. At each such step $t = T+1, \cdots, T+g$ we perform the following operations. First the weighted sunflower transformations are performed on the current hypergraph F_{t-1} and this results in a hypergraph G_t. Then a vertex j (not in B_{t-1} or G_t^1) is chosen, as will be described later. Let $B_t = B_{t-1} \cup \{j\}$ and set the corresponding input bit to 1. Finally, we perform the spreading transformation for this vertex, i.e., the vertex is omitted from each hyperedge containing it, and thus the size of each such hyperedge is decreased by 1.

It is left to show that at every pseudostep t it is possible to choose a new vertex $\{j\}$ to add to B_{t-1}. That means that for any $T+1 \leq t \leq T+g$ we need to show that there is a vertex whose bit can be set to 1 consistently. Note that (I5) holds for the hypergraph G_t also for these pseudosteps and therefore the constraints (edges) that contain vertices in B_{t-1} are redundant. Moreover, the edges in $\cup_{i>1} G_t^i$ that contain vertices from G_t^1 are redundant as well. Thus, each vertex j which is not in $B_{t-1} \cup G_t^1$ can be added to B_{t-1} to continue the process, since the input $\{x_i = 1, i \in B_{t-1} \cup \{j\}$ and $x_i = 0$ otherwise$\}$ is legal in the current hypergraph. Nevertheless, we still have to show that such a vertex j always exists, i.e, $|B_{t-1}| + |G_t^1| < n$. To this end, we first observe that

$$|B_{t-1}| = |B_T| + (t-1-T) < g + g \leq 2n^{1/2}/32 \leq n/16 .$$

Moreover, the potential function increases by at most q at each of these g pseudosteps due to the spreading transformation. Since $gq \leq n/16$ and the potential function was at most $n/4$ at the end of step T, we conclude that during the g pseudosteps $W(F_t) \leq n/4 + n/16$. However, the potential of each edge in $F_t^1 \subseteq F_t$ is 1, and each edge in F_t has a nonnegative potential. Hence,

$$|G_t^1| \leq |F_t^1| \leq W(F_t) \leq n/4 + n/16$$

for the current hypergraph. Therefore

$$|B_{t-1}| + |G_t^1| \leq n/16 + n/4 + n/16 < n$$

and such a j exists. Hence, for any $T < g/k$ we found two inputs with the same history through the T steps, one of which is in L_g and the other of which is not. Thus, we cannot recognize L_g in fewer than g/k steps. This completes the proof of the main theorem. □

4. The complexity of computing symmetric functions. Comparing the relative power of models is known to be an important question. It is known that PRIORITY is strictly stronger than ARBITRARY (see [FMW] and [FRW] for models without a ROM and [LY], [LY1], and [FLRY] for models with a ROM).

The question is whether this is true for symmetric functions. Note that there is a strong connection between the threshold decision problem and computing symmetric functions. In [LY1] it is shown that PRIORITY(1) and ARBITRARY(1), both without ROM, are equivalent for computing symmetric functions of boolean inputs.

Using the results from the previous section with results from [LY2], we extend the results of Li and Yesha and prove that for any polynomial number of processors (in fact,

up to $n^{o(\log n)}$), PRIORITY(m) and ARBITRARY(m), where $m \leq n^{\epsilon}$, both with ROM and the same number of processors, have the same power, up to a factor of at most $n^{2\epsilon}$, for computing any symmetric function. For $m = 1$ we show that the two models are equivalent, up to a constant factor, for a large family of the symmetric functions. In [LY2] it is proved only for a linear number of processors and a smaller class of symmetric functions or for a large enough number of processors.

For any symmetric function f on n bits we associate a function $\overline{f}$ such that $\overline{f}(i) = j$ whenever $f(\overline{x}) = j$ for $|\overline{x}| = i$ ($|\overline{x}| = \sum_{i=1}^{n} x_i$). The threshold of f was defined in [LY2] by

$$|f| = \min\{h + l | \overline{f} \text{ is a constant on the closed interval } [h, n - l]\} .$$

They showed that a lower bound for L_g is also a lower bound for computing a symmetric function f with threshold $|f| = g$. Thus using the result from the previous section we conclude the following theorem.

THEOREM 4.1. *Let f be any symmetric function. For any $\epsilon > 0$* PRIORITY(m) *with* ROM *and a polynomial number of processors requires* $\Omega(\min(\frac{|f|}{m}, \frac{n^{1/2-\epsilon}}{m}))$ *to compute f.*

It is quite easy to design (see [LY2]) an algorithm in ARBITRARY(1) that matches the bound up to a constant factor for $m = 1$ and $|f| \leq n^{1/2-\epsilon}$. Thus for any f and $m \leq n^{\epsilon}$ the bounds are tight up to at most $n^{2\epsilon}$ (in fact, up to n^{δ} for any $\delta > \epsilon$). Moreover, we conclude the following theorem.

THEOREM 4.2. *Let $\epsilon > 0$ be any number and $m \leq n^{\epsilon}$. For any polynomial number of processors,* PRIORITY(m) *and* ARBITRARY(m), *both with* ROM *and the same number of processors, have the same power for computing all symmetric functions on n bits, up to a factor of at most n^{δ} for any $\delta > \epsilon$. For $m = 1$ and $|f| \leq n^{1/2-\epsilon}$ the two models are equivalent up to a constant factor.*

Theorems 4.1 and 4.2 can be extended in the obvious way to the range of super-polynomial number of processors, i.e., when the number of processors is $O(n^d)$, where $d = o(\log n)$.

Acknowledgments. I thank N. Alon for helpful discussions and suggestions that encouraged me to improve the original results. I also thank D. Koller for helpful remarks.

REFERENCES

[Be] P. BEAME, *Lower bounds in parallel machine computation*, Ph.D. thesis, University of Toronto, Toronto, Ontario, 1986.

[FLRY] F. FICH, M. LI, R. RAGDE, AND Y. YESHA, *On the power of concurrent-write* PRAMs *with read-only memory*, Inform. Control, 83 (1989), pp. 234–244.

[FMW] F. FICH, F. MEYER AUF DER HEIDE, AND A. WIGDERSON, *Lower bounds for parallel random access machines with unbounded shared memory*, Adv. Comput. Res., 4 (1987), pp. 1–15.

[FRW] F. FICH, P. RAGDE, AND A. WIGDERSON, *Relation between concurrent write models of parallel computation*, SIAM J. Comput., 17 (1988), pp. 606–627.

[GGKMRS] A. GOTTLIEB, R. GRISHMAN, C. KRUSKAL, K. MCAULIFFE, L. RUDOLF, AND M. SNIR, *The* NYU *Ultracomputer designing a* MIND *shared memory parallel machine*, IEEE Trans. Comput., C-32 (1983), pp. 175–189.

[K] D. KUCK, *A survey of parallel machine organization and programming*, Comput. Surveys, 9 (1977), pp. 29–52.

[LY] M. LI AND Y. YESHA, *Separation and lower bounds for* ROM *and nondeterministic models of parallel computation*, Inform. Control, 73 (1987), pp. 102–128.

[LY1] ———, *New lower bounds for parallel computation*, Proc. 18th ACM Annual Symposium on Theory of Computing, Berkeley, CA, 1986, pp. 177–187.

[LY2] M. Li and Y. Yesha, *Resource bounds for parallel computation of threshold and symmetric functions*, J. Comput. System Sci., 42 (1991), pp. 119–137.

[V] U. Vishkin, *Parallel design space distributed implementation space* (PDDI) *general purpose computer*, Tech. Report RC 9541, IBM T. J. Watson Research Center, Yorktown Heights, NY.

[VW] U. Vishkin and A. Wigderson, *Trade-offs between depth and width in parallel computation*, SIAM J. Comput., 14 (1985), pp. 303–314.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 339–364, April 1992

© 1992 Society for Industrial and Applied Mathematics
010

CONVEX DECOMPOSITION OF POLYHEDRA AND ROBUSTNESS*

CHANDERJIT L. BAJAJ† AND TAMAL K. DEY†

Abstract. This paper presents a simple algorithm to compute a convex decomposition of a nonconvex polyhedron of arbitrary genus (handles) and shells (internal voids). For such a polyhedron S with n edges and r *notches* (features causing nonconvexity in polyhedra), the algorithm produces a worst-case optimal $O(r^2)$ number of convex polyhedra S_i, with $\cup_{i=1}^{k} S_i = S$, in $O(nr^2 + r^{7/2})$ time and $O(nr + r^{5/2})$ space. Recently, Chazelle and Palios have given a fast $O((n + r^2)\log r)$ time and $O(n + r^2)$ space algorithm to tetrahedralize a nonconvex polyhedron. Their algorithm, however, works for a simple polyhedron of genus zero and with no shells (internal voids). The algorithm, presented here, is based on the simple cut and split paradigm of Chazelle. With the help of zone theorems on arrangements, it is shown that this cut and split method is quite efficient. The algorithm is extended to work for a certain class of nonmanifold polyhedra. Also presented is an algorithm for the same problem that uses clever heuristics to overcome the numerical inaccuracies under finite precision arithmetic.

Key words. computational geometry, robust computations, geometric modeling, finite element analysis, computational complexity

AMS(MOS) subject classifications. 68U05, 65Y25, 68Q25

1. Introduction. The main purpose behind decomposition operations is to simplify a problem for complex objects into a number of subproblems dealing with simple objects. In most cases, a decomposition in terms of a finite union of disjoint convex pieces is useful, and this is always possible for polyhedral models [5], [12]. Convex decompositions lead to efficient algorithms, for example, in geometric point location and intersection detection; see [12]. Our motivation stems from the use of geometric models in SHILP,[1] a solid model creation, editing, and display system developed at Purdue [1]. Specifically, a disjoint convex decomposition of simple polyhedra allows for more efficient algorithms in motion planning, in the computation of volumetric properties, and in the finite element solution of partial differential equations.

The surface δS of a polyhedron S is called a 2-manifold if each point on δS has an ϵ-neighborhood that is homeomorphic to an open 2D ball or half-ball [2]. Polyhedra, having 2-manifold surfaces are called manifold polyhedra. Nonmanifold polyhedra may have incidences as illustrated in Fig. 1. Manifold polyhedra with holes are homeomorphic to torii with one or more handles. Manifold polyhedra with *internal voids* are homeomorphic to three-dimensional annuli, that is, spheres with internal voids.

We represent polyhedra with their boundaries, which consist of zero-dimensional faces, called vertices; one-dimensional faces, called edges; and two-dimensional faces, called facets. A *reflex edge* of a polyhedron is an edge where the inner dihedral angle subtended by two incident facets is greater than 180°. Manifold polyhedra can be nonconvex only due to reflex edges. Notches in manifold polyhedra refer to reflex edges only. In nonmanifold polyhedra, however, notches refer to other types of incidences as well; see Fig. 1.

The problem of partitioning a nonconvex polyhedron S into a minimum number of convex parts is known to be NP-hard [22], [24]. Rupert and Seidel [25] also show that

*Received by the editors December 20, 1989; accepted for publication (in revised form) April 1, 1991. A preliminary version of this paper appeared in Proc. Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science 405, Springer-Verlag, 1989, pp. 267–279. This research was supported in part by National Science Foundation grant CCR 90-02228, Office of Naval Research contract N00014-88-K-0402, and Air Force Office of Scientific Research contract 91-0276.

†Department of Computer Science, Purdue University, West Lafayette, Indiana 47907.

[1]SHILP stems from the Sanskrit word SHILP-SHASTRA, the science of sculpture.

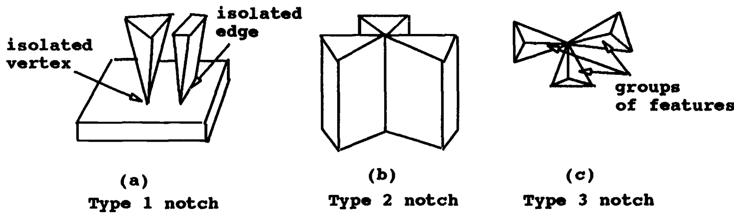


FIG. 1. *Nonmanifold incidences or special notches.*

the problem of determining whether a nonconvex polyhedron can be partitioned into tetrahedra without introducing *Steiner* points is NP-hard. For a given polyhedron S with n edges of which r are reflex, Chazelle [5], [6] established a worst-case $O(r^2)$ lower bound on the number of convex polyhedra needed for complete convex decomposition of S. He gave an algorithm that produces a worst-case optimal number $O(r^2)$ convex polyhedra in $O(nr^3)$ time and in $O(nr^2)$ space. Recently, Chazelle and Palios [7] have given an $O((n + r^2)\log r)$ time and $O(n + r^2)$ space algorithm to tetrahedralize a subclass of nonconvex polyhedra. The allowed polyhedra for their algorithm are all homeomorphic to a 2-sphere, i.e., have no holes (*genus* 0) and shells (internal voids).

Results. In §3, we present an algorithm to compute a disjoint convex decomposition of a manifold polyhedron S that may have an arbitrary number of holes and shells. Given such a polyhedron S with n edges of which r are reflex, the algorithm produces a worst-case optimal $O(r^2)$ number of convex polyhedra S_i with $\cup_{i=1}^{k} S_i = S$ in $O(nr^2 + r^{7/2})$ time and in $O(nr + r^{5/2})$ space. We extend this algorithm to work for nonmanifold polyhedra that do not have abutting edges or facets but may have incidences as illustrated in Fig. 1. The algorithm presented in this paper is based on the repeated cutting and splitting of polyhedra with planes that resolve notches. Chazelle, in [5], first used this method. We improve this method to obtain better time and space bounds based on a refined complexity analysis and the use of efficient algorithms for certain subproblems. In §4, we give an algorithm for the same convex decomposition problem that uses sophisticated heuristics based on geometric reasoning to overcome the inaccuracies involved with finite precision arithmetic computations. This algorithm runs in approximately $O(nr^2 + nr\log n + r^4)$ time and $O(nr + r^{5/2})$ space.

2. Preliminaries.

2.1. Notches. Our algorithm applies to polyhedra that are nonconvex due to the presence of the following four features, called *notches*.

1. *Type* 1 *notches*: These notches are caused by isolated vertices and edges on a facet. An isolated vertex or edge on a facet is not adjacent to any other edge of the facet. See Fig. 1(a).
2. *Type* 2 *notches*: These notches are caused by the edges along which more than two facets meet, as illustrated in the Fig. 1(b). If there are $2k$ ($k > 1$) facets incident on e_i, we assume that they form k notches.
3. *Type* 3 *notches*: These notches are caused by vertices where two or more groups of features (facets, edges) touch each other, as illustrated in the Fig. 1(c). The features within a group are reachable from one another while remaining only on the surface of S and not crossing the vertex. Actually, type 1 notches are a

subclass of these notches. For convenience in the description, we exclude type 1 notches from type 3 notches. The number of groups attached to the vertex determines the number of type 3 notches associated with that vertex.

4. *Type* 4 *notches*: These notches are caused by reflex edges. A manifold polyhedron can have only this type of notches.

The notches of type 1, type 2, and type 3 are called *special notches*, as they are present only in nonmanifold polyhedra. In our algorithm, we first remove all special notches from S, creating only manifold polyhedra. Subsequently, type 4 notches of the manifold polyhedra are removed by repeatedly cutting and splitting the polyhedra with planes resolving the notches. Let an edge g with f_1, f_2 as its incident facets be a notch in a manifold polyhedron. A plane P_g that passes through g is called a *notch plane* if both angles (f_1, P_g) and (P_g, f_2), as measured from the inner side of f_1 and f_2, are not reflex. In other words, a notch plane resolves the reflex angle of a notch. Clearly, for each notch g, there exist infinite choices for P_g. Note that P_g may intersect other notches, thereby producing *subnotches* as well. See Fig. 2.

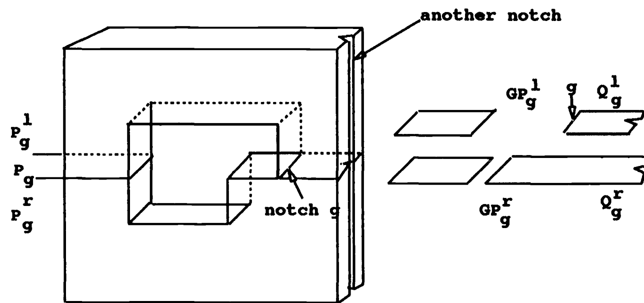


FIG. 2. *A notch and its notch plane, the cross-sectional map, and a cut.*

2.2. Data structure. Let S be a polyhedron, possibly with holes and shells, and having s vertices: $\{v_1, v_2, \cdots, v_s\}$, n edges: $\{e_1, e_2, \cdots, e_n\}$, and q facets: $\{f_1, f_2, \cdots, f_q\}$. These lists of vertices, edges, and facets of S are stored similarly to the *star-edge* representation of polyhedra [19].

Vertices: Each vertex is a record with two fields.

1. *vertex.coordinates*: contains the three-dimensional coordinates of the vertex.
2. *vertex.adjacencies*: contains pointers to the edges incident on the vertex.

Edges: Each edge is a record with two fields.

1. *edge.vertices*: contains pointers to the incident vertices.
2. *edge.orientededges*: contains pointers to the record called *orientededges*, which represent different orientations of an edge on each facet incident on it. The orientation of an edge on a facet f is such that a traversal of the oriented edge has the facet f to its right.

Orientededges: each orientededge is a record with four fields.

1. *orientededge.edge*: contains pointers to the defining edge.
2. *orientededge.facet*: contains pointers to the facet on which the orientededge is incident.

3. *orientededge.orientation*: contains information about the orientation of the edge on the facet.
4. *orientededge.nextorientededge*: contains pointers (possibly more than one) to the next orientededges on the *oriented edge cycle* on a facet. See *facet cycles* below.

Facets: each facet is a record with two fields.

1. *facet.equation*: contains the equation of the plane supporting the facet.
2. *facet.cycles*: contains pointers to a collection of oriented edge cycles bounding the facet. The traversal of each oriented edge cycle always has the facet to the right. Each oriented edge cycle is represented with a linked list of orientededges on the cycle. If there is an isolated vertex on the facet (Fig. 1(a)) a pointer to the vertex is included in *facet.cycles* as a degenerate oriented edge cycle. An isolated edge is represented with the oriented edge cycle of two orientededges. For a nonmanifold polyhedron, a facet may have configurations as shown in Fig. 3 where a vertex or an edge is considered more than once in the oriented edge cycles, though an oriented edge is included only once.

2.3. Useful lemmas. Let the *polygonal boundary* refer to an oriented edge cycle embedded on a plane with no edge intersecting the other except at their endpoints. The traversal of a polygonal boundary may pass through an edge or a vertex more than once. In the rest of the paper, we use the term polygon to mean a *connected* region on a plane that is bounded by one or more polygonal boundaries. For example, such a polygon corresponding to the facet f is shown in Fig. 3. Let G be a polygon with vertices $v_1, v_2, \cdots, v_k$ in clockwise order. A vertex v_i is a *reflex vertex* of G if the outer angle between the oriented edges $d_{i-1} = (v_{i-1}, v_i)$ and $d_i = (v_i, v_{i+1})$ is less than or equal to 180°. The outer angle between two consecutive oriented edges d_{i-1} and d_i is measured in the anticlockwise direction from d_i to d_{i-1}. Note that with this definition, v_4, v_5 of the nonsimple facet in Fig. 3 are reflex vertices, though v_3 is not. The vertices that are not reflex vertices are called *normal vertices* of G. The boundary of a polygon G can be partitioned into x-monotone (or y-monotone) maximal pieces called *monotone chains*, i.e., vertices of a monotone chain have x-coordinates (or y-coordinates) in either strictly increasing or decreasing order. See Fig. 4.

In subsequent sections, we use the following lemmas.

LEMMA 2.1. *Let G be a polygon with r reflex vertices. The number of monotone chains c in G is bounded as $c \leq 6(1 + r)$.*

Proof. The proof follows from Theorem 3, [5, p. 22]. □

LEMMA 2.2. *Let G be a polygon with s normal vertices. There are at most $O(s)$ monotone chains in G.*

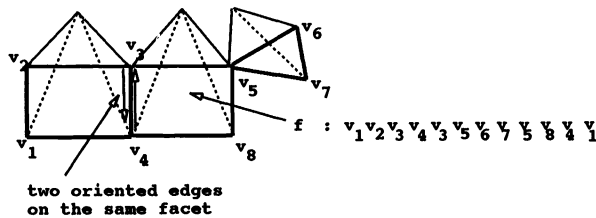


FIG. 3. *A nonsimple facet.*

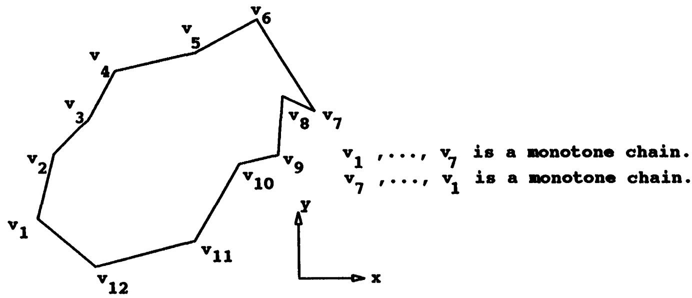


FIG. 4. *Monotone chains in a polygon.*

Proof. Let v be the vertex of G with the minimum y-abscissa, and let B be the boundary obtained by removing the vertex v and an ϵ*-ball* around v from the boundary of G. Add six more edges to B, as shown in Fig. 5, to construct a new polygon G'. The polygon G' is oppositely oriented with respect to G. Note that each reflex vertex of G' corresponds to a normal vertex of G. Thus G' has no more than s reflex vertices, and according to Lemma 2.1, its boundary is partitioned into $O(s)$ monotone chains. The polygon G cannot have more monotone chains than G', which implies that G has $O(s)$ monotone chains. □

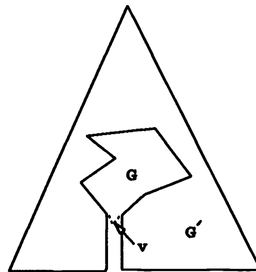


FIG. 5. *Constructing a polygon of opposite orientation.*

In the following lemma, the line segments of a line that are interior to a polygon are called *chords*.

LEMMA 2.3. *Let G be a polygon (possibly with holes) with r reflex vertices. No line can intersect G in more than $r + 1$ chords.*

Proof. The proof proceeds inductively. The case for $r = 0$ is trivial. In the general step, consider a polygon G with $r = k \geq 1$ reflex vertices. Take an arbitrary reflex vertex, and resolve it by a cut through it. The cut may separate G into two polygons G_1 and G_2 of r_1 and r_2 reflex vertices, respectively, such that $r_1 + r_2 \leq k - 1$. Furthermore, the number of chords of a line L in G cannot exceed the sum of the number of chords in G_1 and G_2. Therefore, using the induction hypothesis, one can conclude that the line L intersects G in no more than $r_1 + 1 + r_2 + 1 \leq k + 1$ chords. If, however, the cut does

not split G, one ends up with a polygon G' of at most $k-1$ reflex vertices. Since the line L may intersect the cut, just performed, the number of chords in G is less than or equal to that in G', which again implies that the former is less than or equal to $k-1+1 \leq k+1$. □

LEMMA 2.4. *Let $\wp$ be a set of k polygons with r reflex vertices. No line can intersect $\wp$ in more than $r+k$ chords.*

Proof. The proof follows immediately from Lemma 2.3. □

2.4. Nesting of polygons. The following *polygon nesting* problem arises as a subproblem in our polyhedral decomposition. Let $\wp$ be a set of k polygons $G_i, i = 1, \cdots, k$, none of which intersects others along its boundary. Corresponding to each polygon G_i, we define *ancestor*(G_i) as the set of polygons containing G_i. The polygon G_k in ancestor(G_i) is called the *parent* of G_i if ancestor(G_k)=ancestor$(G_i) - G_k$. Note that there may not exist any such G_k, since ancestor(G_i) may be empty. In that case, we say that the parent of G_i is *null*. Any polygon with parent G_k is called the *child* of G_k. In Fig. 6, ancestor$(G_3) = \{G_1, G_2\}$, parent$(G_4) = (G_2)$, children$(G_2) = \{G_3, G_4\}$, ancestor(G_5) =null=children(G_5). The *nesting structure* of $\wp$ is an acyclic directed graph (a forest of trees) in which there is a node n_i corresponding to each polygon G_i in $\wp$, and a directed edge from a node n_i to n_j if and only if G_j is the parent of G_i. The polygon nesting problem is to compute the nesting structure of a set of nonintersecting polygons.

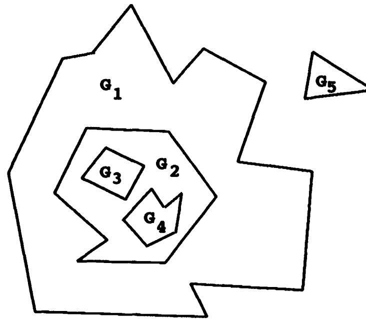


FIG. 6. *Nested polygons.*

LEMMA 2.5. *The problem of polygon nesting for a set of nonintersecting polygons can be solved in $O(s + t\log t)$ time assuming exact numerical computations where s is the total number of vertices, and t is the total number of monotone chains present in all input polygons.*

Proof. See [4]. Though the algorithm given in [4] uses a slightly different type of monotone chains, called *subchains*, it also works for the monotone chains as defined in this paper. Further, the algorithm of [4] can be straightforwardly adapted to the input set of polygons as defined in this paper. □

3. Convex decomposition.

3.1. Sketch of the algorithm. Given a polyhedron S, it is first split along the vertices and edges of special notches to produce manifold polyhedra. Reflex edges of a manifold polyhedron are removed by slicing it with notch planes. Notch planes may possibly intersect other notches to create subnotches. In general, the notch elimination process produces a number of subpolyhedra. At a generic step of the algorithm, all subnotches of a notch, present in possibly different subpolyhedra, are eliminated with a single notch plane. Slicing a manifold polyhedron with a plane may produce nonmanifold subpolyhedra with special notches. See Fig. 7. As before, these nonmanifold subpolyhedra are split along the special notches to produce only manifold polyhedra. If the notch plane, however, does not pass through a vertex of the polyhedron being cut, manifold property is preserved in the resulting subpolyhedra.

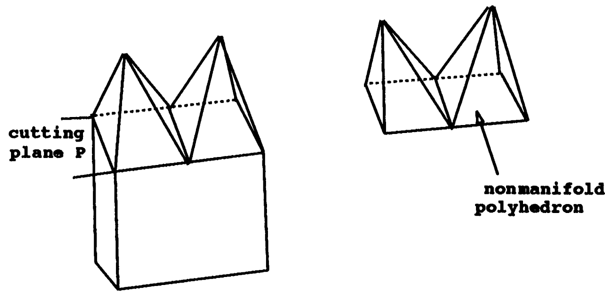


FIG. 7. *An example where the manifold property is not preserved after a cut.*

Algorithm ConvDecomp(S)
Step 1: Remove all special notches from S. This produces manifold polyhedra.
Step 2: Assign a notch plane for each notch in the manifold polyhedra produced in Step 1.
Step 3: repeat
    Let $g_1, g_2, \cdots, g_k$ be the subnotches of a notch g present in the polyhedra $S_1, S_2, \cdots, S_k$. Let P_g be the notch plane assigned to g. Remove $g_1, g_2, \cdots, g_k$ from $S_1, S_2, \cdots, S_k$ by the notch plane P_g.
    Remove special notches produced by this slicing operation.
  until all notches are eliminated.
end.

Step 1 of the algorithm is described in §3.3. Step 2 can be performed trivially in $O(r)$ time. The slicing step of the algorithm (Step 3) needs to be performed carefully and is detailed below in §3.2.

3.2. Intersecting a manifold polyhedron with a notch plane. Let S be a manifold polyhedron with r notches and p edges. By S, we denote any polyhedron $S_1, S_2, \cdots, S_k$ that is encountered in Step 3 of the above algorithm ConvDecomp. The notch plane P_g: $ax + by + cz + d = 0$ defines two closed half-spaces $P_g^\ell : ax + by + cz + d \geq 0$ and $P_g^r : ax + by + cz + d \leq 0$. To cut a polyhedron S with the plane P_g, it is essential to

compute

$$S^\ell = \mathrm{cl}(\mathrm{int}(P_g^\ell) \cap \mathrm{int}(S)),$$
$$S^r = \mathrm{cl}(\mathrm{int}(P_g^r) \cap \mathrm{int}(S)),$$

where $\mathrm{cl}(O)$ and $\mathrm{int}(O)$ denote the closure and interior of the geometric object O. Since polyhedra are represented with their boundaries, we need to compute the boundaries δS^ℓ and δS^r of S^ℓ and S^r, respectively. To compute δS^ℓ and δS^r, it is essential to compute the features of δS^ℓ and δS^r lying on P_g, which are given by

$$GP_g^l = P_g \cap \delta S^\ell,$$
$$GP_g^r = P_g \cap \delta S^r.$$

We refer to GP_g^ℓ and GP_g^r as *cross-sectional maps*. Note that for a polyhedron S and a plane P_g, the cross-sectional maps GP_g^ℓ and GP_g^r may be different. See, for example, Fig. 2. In general, GP_g^ℓ and GP_g^r consist of a set of isolated points, segments, and polygons, possibly with holes. The unique polygons Q_g^ℓ and Q_g^r on GP_g^ℓ and GP_g^r, respectively, containing the notch g on their boundary, are called *cuts*. Note that to remove a notch g, it is sufficient to slice S along only the cut instead of the entire cross-sectional map.

Instead of computing Q_g^ℓ and Q_g^r separately, we first compute the cut $Q_g = Q_g^\ell \cup Q_g^r$ and then refine it to obtain Q_g^ℓ and Q_g^r. This calls for computing the cross-sectional map $GP_g = GP_g^\ell \cup GP_g^r$. The polygon corresponding to the cut Q_g may have a vertex or an edge appearing more than once while traversing its boundary. If an edge appears more than once in traversing the boundary of Q_g^ℓ or Q_g^r, the edge must make the corresponding subpolyhedron nonmanifold. See Fig. 7. It is interesting to observe that there can be at most four facets incident upon that edge since the original polyhedron being sliced was a manifold.

An additional fact is that a single slicing along the cut may not separate the polyhedron S into two different pieces; see Fig. 2. In this case, two facets corresponding to Q_g^ℓ and Q_g^r are created that may overlap geometrically and be considered distinct, so that the polyhedron is treated as manifold polyhedron.

The algorithm to cut a polyhedron S with a notch plane P_g consists of two basic steps.

- *Step* I: Computing the cut Q_g: This calls for computing inner (holes) and outer boundaries of the polygon Q_g.
- *Step* II: Splitting the polyhedron S.

Step I is detailed below in §3.2.1 and Step II in §3.2.2.

3.2.1. Computation of the cut Q_g. *Step* A. First, all boundaries present in the cross-sectional map GP_g are computed. To do this, all the facets of S are visited in turn. If the notch plane intersects a facet f, all intersection points are computed. Note that f must be a simple facet (no vertex or edge is traversed twice along its boundaries) since S is a manifold polyhedron. Let $a_1, a_2, \cdots, a_k$ be the sorted sequence of intersection points along the line of intersection $P_g \cap f$. We call an intersection point a *new intersection vertex* if it does not coincide with any vertex of the facet f and we call it an *old intersection vertex* otherwise. It is essential to decide consistently whether there should be an edge between two consecutive intersection vertices a_i and a_{i+1} of this sorted sequence. This is done by

scanning the vertices in sorted order and deciding whether we are "inside" or "outside" the facet as we leave a vertex to go to the next one. If a_i is a new intersection vertex, there can be an edge between a_i and a_{i+1} only if there is no edge between a_{i-1} and a_i and vice versa. On the other hand, if a_i is an old intersection vertex, there can be an edge between a_i and a_{i+1} irrespective of the presence of an edge between a_{i-1}, a_i.

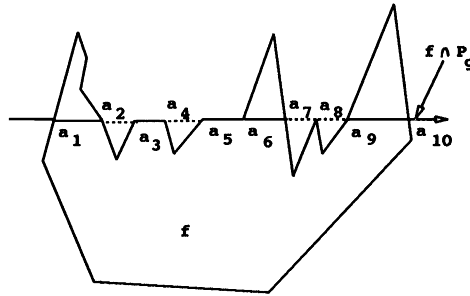


FIG. 8. *Generating new and old edges.*

Switching between "inside" and "outside" of the facet is carried out properly, even with degeneracies, using a *multiplicity code* at each intersection vertex. During the scan of the sorted sequence of intersection vertices, a counter is maintained. The counter is initialized to zero and is incremented by the multiplicity code at each vertex. Our status toggles between "inside" and "outside" of the facet as the counter toggles between the "odd" and "even" count. A new intersection vertex is assigned a multiplicity code of 1. An old intersection vertex has a multiplicity code of 1 if both of its incident oriented edges on the facet f do not lie in the same half-space of P_g and a multiplicity code of 2 otherwise. If there is an old edge (edge of f) between two vertices a_i and a_{i+1}, multiplicity codes are assigned to them as follows. If another two incident oriented edges on a_i, a_{i+1} on the facet f lie in the same open half-space of the notch plane, assign a multiplicity code of 1 to both of them. Otherwise, assign multiplicity codes of 1 and 2 to a_i and a_{i+1} in any order. In Fig. 8, there is an old edge between a_3 and a_4. The status ("outside") with which one enters the vertex a_3 is same as the one with which one leaves the vertex a_4. This is enforced by assigning a multiplicity code of 1 on the two vertices that increments the counter by an "even" amount and prevents it from toggling. In the same example, there is another old edge between a_5 and a_6. The status ("outside") with which one enters the vertex a_5 is different from the one with which one leaves the vertex a_6. This is enforced by assigning multiplicity codes of 1 and 2 on the two vertices in any order which increment the counter by an "odd" amount and make it toggle. A new edge from the vertex a_i to a_{i+1} is created if the count is "odd" on leaving the vertex a_i. In case there is an old edge between a_i and a_{i+1}, no new edge is created between them. This process is repeated for all facets intersected by P_g resulting eventually in creating the 1-*skeleton* or the underlying graph of GP_g. This underlying graph becomes a directed graph if the oriented edges associated with the edges in GP_g are considered. Orientation of each such edge is determined in constant time since the orientations of the facets intersecting the notch plane are known. A traversal in a depth-first manner in this directed graph traces the boundaries of GP_g.

Timing analysis. According to Lemma 2.3, the notch plane P_g intersects a facet f of S in at most $2r_i + 2$ points where r_i is the number of reflex vertices in f. Thus, sorting of the intersection points on a facet takes at most $O(u_i \log r_i)$ time where u_i is the number of intersection points on the facet. Considering all such facets, we obtain the sorted sequence of intersection vertices on the facets computed in $O(p+u \log r)$ time, where u is the number of vertices in GP_g. Generating the edges between these intersection vertices takes no more than $O(p)$ time altogether. The time taken for tracing the boundaries of GP_g is linear in the number of edges in GP_g. Overall, the computation of GP_g takes $O(p + u \log r)$ time.

Step B. Next, the inner and outer boundaries of Q_g are determined from GP_g. It is trivial to determine the boundary B_g containing the notch g. One can determine whether B_g is an inner or outer boundary of Q_g by checking the orientations of the edges on the boundary.

Case(i). B_g is an outer boundary of Q_g. Let I_i be the polygon corresponding to an inner boundary (hole) of Q_g. The polygon I_i has at least one vertex that is normal. Since the boundary of I_i constitutes an inner boundary of Q_g, the normal vertices of I_i are reflex vertices of Q_g. Definitely, reflex vertices of Q_g lie on notches of S. This implies that all inner boundaries of Q_g will have a vertex where P_g intersects a notch of S. The set W of boundaries having at least one such vertex is determined. The boundaries in the set $W \cup B_g$ are called *interesting boundaries*. The polygon nesting algorithm applied on the polygons constituted by the interesting boundaries detects the children of B_g. The boundaries of these children constitute the inner boundaries of Q_g.

Timing analysis. The set W can be created in $O(u)$ time where u is the number of vertices present in the cross-sectional map. Certainly, the number of interesting boundaries is $O(t)$ where t is the number of notches intersected by the notch plane P_g. The interesting boundaries that are outer boundaries of some polygon in the cross-sectional map have $O(t)$ reflex vertices, since these vertices are generated by the intersection of a notch of S with the notch plane. On the other hand, the interesting boundaries that are inner boundaries of some polygon in the cross-sectional map have $O(t)$ normal vertices. Thus, according to Lemmas 2.1 and 2.2, there are at most $O(t)$ monotone chains in the interesting boundaries. If there are u' vertices in the interesting boundaries, the children of B_g can be determined in $O(u' + t \log t)$ time using the polygon nesting algorithm (Lemma 2.5). Thus, in this case, the inner and outer boundaries of Q_g can be detected in $O(u + u' + t \log t) = O(p + t \log t)$ time, since $u' = O(u) = O(p)$.

Case(ii). B_g is an inner boundary of Q_g. The boundaries that completely contain the boundary B_g inside are determined. This can be done by checking the containment of any point on B_g with respect to all boundaries in the cross-sectional map. These boundaries, together with B_g, are the interesting boundaries. The polygon nesting algorithm, applied on these interesting boundaries, detects the boundaries of the parent polygon of B_g. This boundary is the outer boundary of Q_g. Note that Q_g may have other inner boundaries different from B_g. Once the outer boundary of Q_g is computed, all of its inner boundaries can be obtained applying the technique used in Case (i).

Timing analysis. Detection of all boundaries containing B_g takes $O(u)$ time. The set of interesting boundaries can be partitioned into two classes according to whether they are inner or outer boundaries of some polygon. It is not hard to see that there can be at most one more outer boundary than inner boundaries in this set. Hence, the number of interesting boundaries is of the order of inner boundaries present in the cross-sectional map. As discussed in Case (i), the number of inner boundaries must be bounded above by the number of notches intersected by the notch plane. Thus, there are $O(t)$ interesting

boundaries. Further, as explained before, the number of monotone chains present in these interesting boundaries can be at most $O(t)$. Hence, the outer boundary of Q_g can be determined in $O(p + t \log t)$ time. Detection of other inner boundaries that are different from B_g takes another $O(p + t \log t)$ time. Thus, in this case also all outer and inner boundaries of Q_g can be detected in $O(p + t \log t)$ time.

Combining all these costs together, we see that the "cut computation" takes $O(p + t \log t + u \log r)$ time.

3.2.2. Splitting S. Separation of S along the cut Q_g is carried out by splitting facets that are intersected by Q_g. Suppose f is such a facet, which is to be split at $a_1, a_2, \cdots, a_k$. The splitting of f consists of splitting the edges on which a new intersection vertex lies and the old intersection vertices. For this splitting operation, the intersection vertices on each facet f are visited and for each such intersection vertex, constant time is spent for setting relevant pointers. The facet f may be split into several subfacets. The inner boundaries of f that are not intersected by P_g remain as inner boundaries of some of these subfacets. The polygon nesting algorithm determines the inclusions of these inner boundaries into proper subfacets. The cut Q_g is refined to yield Q_g^ℓ and Q_g^r. It is observed that the differences between Q_g^ℓ and Q_g^r are caused by the edges of S that lie completely on P_g. Hence, to refine Q_g, one needs to determine which of the edges of S are to be transferred to Q_g^ℓ (Q_g^r, respectively). This can be done using the following simple rule. An old edge e must be transferred to Q_g^ℓ (Q_g^r, respectively) if any facet (or a part of it) that is adjacent to e and not coplanar with P_g lies in P_g^ℓ (P_g^r, respectively). A copy of Q_g is created and one of the two Q_g's is designated for Q_g^ℓ and another for Q_g^r. From a copy, all those edges that are not to be transferred to it are deleted. Note that the transfer of edges lying on Q_g takes care of the facets lying on Q_g. Two oppositely oriented facets at the same geometric location corresponding to the cuts Q_g^ℓ and Q_g^r are created. All modified incidences are adjusted properly. A depth-first traversal in the modified vertex list either completes the separation of S by collecting all the pertinent features of each piece or reveals the fact that S is not separated into two different pieces by the cut. In the latter case, either the number of holes or the number of shells in S is reduced by one.

Timing analysis. Adjustment of all incidences in the internal structure of S cannot take more than $O(p)$ time since each edge is visited only $O(1)$ times. The polygon nesting takes $O(p + r \log r)$ time since there can be at most $O(r)$ holes in the facets of S containing $O(r)$ monotone chains. Further, creation of Q_g^ℓ and Q_g^r from Q_g and the depth-first traversal in the modified vertex list cannot exceed $O(p)$ time. Hence, the "splitting operation" takes $O(p + r \log r)$ time.

3.3. Elimination of special notches and its analysis. For a nonmanifold polyhedron S, nonconvexity results from four types of notches, as discussed in §2.1. Let S have n edges and r notches. The counting of special notches is described in §2.1. A preprocessing is carried out as follows to remove the notches of the first three types, called *special notches*.

Removal of type 1 *notches.* As can be observed from Fig. 1(a), the vertex or the edge causing the nonconvexity is detached from the facet on which it is incident as an isolated vertex or an isolated edge. Identifying these vertices and edges and detaching them from the corresponding facets take at most $O(n)$ time.

Removal of type 2 *notches.* Here, more than two facets are incident on an edge e_i. Let these facets be $f_1, f_2, \cdots, f_{r_i}$. Let C be a cross-section obtained as the intersection of the facets incident on e_i with the plane P that is normal to the edge e_i. C consists of edges $e_j = (f_j \cap P)$. The facets around e_i are sorted circularly by a simple circular sort

of the edges e_j's around $e_i \cap P$. The adjacent facets that enclose a volume of S are paired. Let this pairing be $(f_1, f_2), (f_3, f_4), \cdots, (f_{r_i-1}, f_{r_i})$. An edge between each pair of facets is created and the edge e_i is deleted. All these edges are at the same geometric location of e_i. All incidences are adjusted properly. Sorting of facets around the edge e_i takes $O(r_i \log r_i)$ time. Further, for all type 2 notches, the adjustment time of all incidences in the internal representation of S cannot exceed $O(n)$. Thus, the removal of all type 2 notches takes at most $(n + r \log r)$ time.

Removal of type 3 *notches*. Let v be a vertex that corresponds to a type 3 notch. In this case, we group together all features (edges and facets) that are incident on v and are reachable from one another while remaining always on the surface of S and never crossing v. This gives a partition of the features incident on v into smaller groups. For each such group, a vertex at the same geometric location of v is created and all incidences are adjusted properly. This, in effect, removes the nonconvexity caused by v. All such vertices causing type 3 notches in S can be identified in $O(n)$ time by edge-facet-edge traversal on the internal data structure of S. Removal of all such notches takes at most $O(n)$ time. This is due to the fact that each edge can be adjacent to at most two type 3 notches and thus is visited only $O(1)$ times. Thus, all type 3 notches can be removed in $O(n)$ time.

Finally, a mixture of cases may occur where an isolated vertex is also a type 3 notch or an isolated edge is also a type 2 notch. All these cases are handled by first eliminating all type 1 notches and then eliminating type 3 notches followed by type 2 notches.

Removal of all the above notches generates at most $O(n)$ new edges and produces at most k manifold polyhedra where k is the number of special notches in S.

3.4. Worst-case complexity analysis. Combining the costs of the "cut computation" of §3.2.1 and the "splitting operation" of §3.2.2 yields the following lemma.

LEMMA 3.1. *A manifold polyhedron S having p edges can be partitioned with a notch plane P_g of a notch g in $O(p + t \log t + (u + r) \log r)$ time and in $O(p)$ space where t is the number of notches intersected by P_g, and u is the number of vertices in GP_g.*

The following two-dimensional subproblem is essential for the analysis of ConvDecomp. Let L be a set of r lines in two-dimensions that form a line arrangement A[12]. Let E be a set of edges removed from A such that all cells in $A - E$ are convex. Let us denote the new arrangement $A - E$ as A^-. Let C be a set of cells in A^- intersected by a line l. The total number of edges in the cells in C determines the zone complexity $z(l, A^-, r)$ of l in A^-. Of course, the contribution of a line in any single cell is counted only once, although it may have several consecutive segments on it in that cell. Let $q(r) = \max\{z(l, A^-, r) | l \text{ is any line in any such arrangement } A^-\}$. In Lemma 3.2 below, we derive a nontrivial upper bound for $q(r)$. Now suppose that a polyhedron S with n edges and r notches has been sliced with $y \leq r$ notch planes so far. Let $S_1, S_2, \cdots, S_k$ be the polyhedra in the current decomposition, where each S_i contains a subnotch g_i of a notch g in S. Let x_i be the number of edges on Q_{g_i}.

LEMMA 3.2. $x = \sum_{i=1}^{k} x_i = O(n + r^{3/2})$.

Proof. Consider the cut Q_g produced by the intersection of S with P_g. The region in Q_g is divided into smaller cells by the segments of *notch lines* produced by the intersection of other notch planes with P_g. It is important to note that consecutive segments of a notch line may have gaps in them. We focus on the cells $Q_{g_1}, Q_{g_2}, \cdots, Q_{g_k}$ adjacent to the subnotches $g_1, g_2, \cdots, g_k$ of the notch g.

Consider separately the set of notch line segments that divides Q_g. These line segments and the line L_g corresponding to the notch g produce an arrangement T of line segments on the notch plane P_g. Notice that the arrangement T can be thought of as

an arrangement A^- for some arrangement A of y lines. The cells adjacent to the line L_g in this arrangement form the *zone* Z_g of L_g. Let the set of vertices and edges of Z_g be denoted as V_g and E_g, respectively. Note that in each single cell of Z_g, consecutive segments of a line form a single edge. Actually one can verify that this notion of edges is consistent with our notion of cuts. Overlaying Q_g on T produces $Q_{g_1}, Q_{g_2}, \cdots, Q_{g_k}$. See Fig. 9. These are the cells in $T \cup Q_g$ that are adjacent to the line L_g. Let V'_g and E'_g denote the sets of vertices and edges, respectively, in $Q_{g_1}, Q_{g_2}, \cdots, Q_{g_k}$. The vertices in V'_g can be partitioned into three disjoint sets, namely, T_1, T_2, T_3. The set T_1 consists of vertices formed by the intersections of two notch line segments; T_2 consists of vertices of Q_g, and T_3 consists of vertices formed by the intersections of the notch line segments with the edges of Q_g. Certainly, $|T_1| = O(|E_g|) = O(q(y))$. If Q_g has u' vertices, $|T_2| \leq u'$.

To count the number of vertices in T_3, we first assume that Q_g does not have any holes. Consider an edge e in E_g that contributes one or more edge segments to E'_g as a result of intersections with Q_g. There must be at least one reflex vertex of Q_g present between two such successive edge segments of e. Charge one unit cost to the reflex vertex that lies to the left (or right) of each segment, and charge one unit cost to e itself for the leftmost (or rightmost) segment. We claim that each reflex vertex of Q_g is charged at most once by this method. Suppose, on the contrary, a reflex vertex is charged twice by this procedure. That reflex vertex must appear between two segments of two edges in E_g, as shown in Fig. 9(b). As can be easily observed, all four edge segments cannot be adjacent to the regions incident on the edge g of Q_g. This contradicts our assumption that all these four edge segments are present in E'_g. Hence the total charge incurred upon the reflex vertices of Q_g and the edges of E_g can be at most $O(r_g + q(y))$, where r_g is the number of reflex vertices present in Q_g. This implies that as a result of intersections with Q_g, at most $O(r_g + q(y))$ segments of edges in E_g contribute to E'_g. Hence $|T_3| = O(r_g + q(y))$.

Consider next the case where Q_g has holes. We refer to the polygon corresponding to a hole in Q_g as a *hole-polygon*. From Q_g create a polygon Q'_g that does not have any hole, by merging all polygons into a single polygon as follows. Let H_1 and H_2 be two hole-polygons that have at least two visible vertices v_1, v_2, i.e., the line segment joining v_1, v_2 does not intersect any other edge. Split v_1 and v_2 and join them with the line segments, as shown in Fig. 10 to merge H_1, H_2. Repeat this process successively for all hole-polygons until they are merged into a single polygon. Finally, connect the boundary of this new polygon to the outer boundary of Q_g to create Q'_g. Consider superimposing

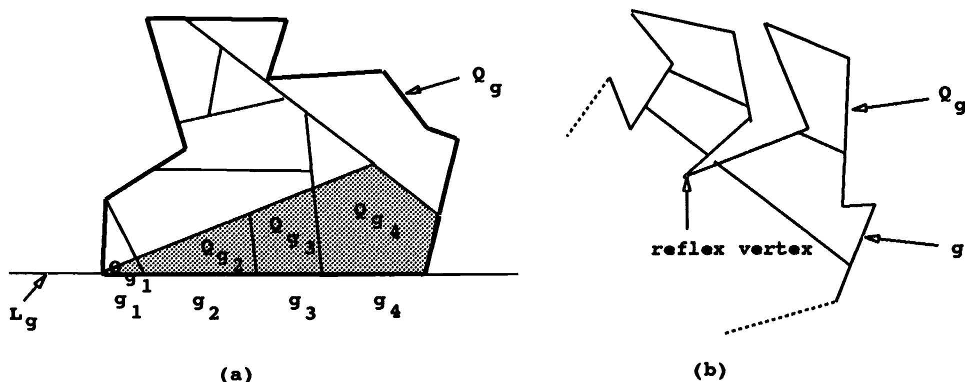


FIG. 9. *Superimposing a cut on an arrangement of notch line segments.*

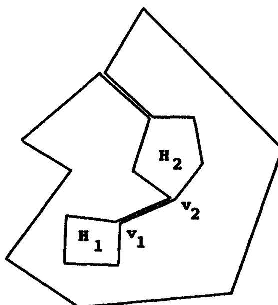


FIG. 10. *Merging polygons to create Q'_g from Q_g.*

Q'_g on Z_g. Let T'_3 denote the set of vertices formed by the intersection of edges of E_g and those of Q'_g. The distance between split vertices of Q'_g can be kept arbitrarily small to preserve all intersections between the edges of Q_g and those of Z_g. This ensures that $|T_3| \leq |T'_3|$. The polygon Q'_g has at most $O(u')$ vertices since the original polygon Q_g had u' vertices, and at most $O(u')$ extra vertices are added to form Q'_g from Q_g. Furthermore, the polygon Q'_g can have at most $O(u')$ reflex vertices. Applying the previous argument on the superimposition of Q'_g on Z_g, we get $|T_3| \leq |T'_3| = O(u' + q(y))$.

Putting all these together, we have $|V'_g| = |T_1| + |T_2| + |T_3| = O(r_g + q(y) + u')$. Certainly, $r_g \leq r$, $y \leq r$, and $u' \leq n$. This gives $|V'_g| = O(n + q(r))$. Since $Q_{g_1}, Q_{g_2}, \cdots, Q_{g_k}$ form a planar graph, we have $x = |E'_g| = O(|V'_g|) = O(n + q(r))$. Now we show that $q(r) = O(r^{3/2})$, which completes the proof.

Let C be the set of cells intersected by a line in an arrangement A^- formed out of an arrangement A of r lines. Form a bipartite graph $G = (V_1 \cup V_2, E)$, where each node in V_1 corresponds to a cell in C and each node in V_2 corresponds to a line in A. An edge $e \in E$ connects two vertices $v_1 \in V_1$, $v_2 \in V_2$ if the line corresponding to v_2 contributes an edge to the cell corresponding to v_1. Observe that any four lines in A can contribute simultaneously to at most two cells in C since they are convex [12]. This means that G cannot have $K_{2,5}$[2] as a subgraph. Then using results from forbidden graph theory [20], G can have at most $O(cr^{1/2} + r)$ edges, where $|C| = c$. Since $|C| \leq r + 1$, we have $q(r) = O(|E|) = O(r^{3/2})$. □

LEMMA 3.3. *The total number of edges in the final decomposition of a polyhedron S with r notches and n edges is $O(nr + r^{5/2})$.*

Proof. Edges in the final decomposition consist of newly generated edges by the cuts, and the edges of S that are not intersected by any notch plane. By Lemma 3.2, the total number of edges present in all cuts corresponding to the subnotches of a notch is $O(n + r^{3/2})$. This implies that each notch plane generates $O(n + r^{3/2})$ new edges. Thus, r notch planes generate $O(nr + r^{5/2})$ new edges. Hence, the total number of edges in the final decomposition is $O(n + nr + r^{5/2}) = O(nr + r^{5/2})$. □

LEMMA 3.4. *Let $S_1, S_2, \cdots, S_k$ be the polyhedra in the current decomposition, where*

[2]It is a complete bipartite graph $G = (V_1 \cup V_2, E)$ where $|V_1| = 2$ and $|V_2| = 5$.

Proof. Consider the cross-sectional map GP_g. The lines of intersection between P_g and other notch planes, called the notch lines, divide this map into smaller facets. These facets are present in the cross-sectional maps in $S_1, S_2, \cdots, S_k$, i.e., in $\cup_{i=1}^{k} GP_{g_i}$. The vertices in $\cup_{i=1}^{k} GP_{g_i}$ can be partitioned into three sets, viz., T_1, T_2, and T_3. The set T_1 consists of vertices that are created by intersections of two notch lines. The set T_2 consists of vertices of GP_g, and the set T_3 consists of vertices that are created by intersections of edges of GP_g and notch lines . Since there are at most r notch lines, $|T_1| = O(r^2)$. Certainly, $|T_2| = O(n)$. By Lemma 2.4, each notch line can intersect GP_g in at most $O(r)$ chords, since GP_g can have at most r polygons containing no more than r reflex vertices altogether. This gives $|T_3| = O(r^2)$. Thus,

$$u = \sum_{i=1}^{k} u_i = |T_1| + |T_2| + |T_3|$$
$$= O(n + r^2). \qquad \square$$

As discussed in [6], one can always produce a worst-case optimal number ($O(r^2)$) of convex polyhedra by carefully choosing the notch planes.

LEMMA 3.5. *A manifold polyhedron S with r notches can be decomposed into $\frac{r^2}{2} + \frac{r}{2} + 1$ convex pieces if all subnotches of a notch are eliminated by a single notch plane. Further, this convex decomposition is worst-case optimal since there exists a class of polyhedra that cannot be decomposed into fewer than $O(r^2)$ convex pieces.*

Proof. See [6] for the proof. $\square$

THEOREM 3.1. *A manifold polyhedron S, possibly with holes and shells and having r notches and n edges, can be decomposed into $O(r^2)$ convex polyhedra in $O(nr^2 + r^{7/2})$ time and $O(nr + r^{5/2})$ space.*

Proof. Decomposition of a polyhedron consists of a sequence of cuts through the notches of S, as illustrated in the algorithm ConvDecomp. Step 1 assigns a notch plane for each notch in S in $O(r)$ time. According to Lemma 3.5, ConvDecomp produces worst-case optimal $O(r^2)$ convex pieces at the end since all subnotches of a notch are removed by a single notch plane. Note that all holes and shells are removed automatically by the notch elimination process.

At a generic instance of the algorithm let S_1, $S_2, \cdots, S_k$ be k distinct (nonconvex) polyhedra in the current decomposition, where each S_i contains a subnotch g_i of a notch g that is going to be removed. Let S_i have m_i edges of which r_i are notches. Let t_i be the number of notches intersected by P_g in S_i and $t = \sum_{i=1}^{k} t_i$ and u_i be the number of vertices in GP_{g_i} of S_i and $u = \sum_{i=1}^{k} u_i$.

Applying Lemma 3.1, removal of a notch g can be carried out in $O(\sum_{i=1}^{k}(m_i + t_i \log t_i + (u_i + r_i)\log r_i))$ time. Since $m = \sum_{i=1}^{k} m_i = O(nr + r^{5/2})$, $\sum_{i=1}^{k} r_i = O(r^2)$, $u = O(n+r^2)$, and since a notch plane can intersect at most $r-1$ notches giving $t = O(r)$, we have $O(\sum_{i=1}^{k}(m_i + t_i \log t_i + (u_i + r_i)\log r_i)) = O(nr + r^{5/2})$.

As described before, elimination of a notch may produce nonmanifold polyhedra having special notches. To remove them, the same method is used for eliminating special notches as used for the original polyhedron. Note that the type 2 notches in these nonmanifold polyhedra can be adjacent to at most four facets. Hence, no logarithmic factor appears in the time complexity of removing such notches. This implies that the elimination of special notches from the nonmanifold polyhedra produced as a result of cutting *each S_i contains a subnotch g_i of a notch g. Let u_i be the total number of vertices in the cross-sectional map in S_i. Then we have $u = \sum_{i=1}^{k} u_i = O(n + r^2)$ where u is the total number of vertices in the cross-sectional maps in $S_1, S_2, \cdots, S_k$.*

manifold polyhedra with notch planes can be carried out in totally $O(m) = O(nr + r^{5/2})$ time.

Thus, each notch elimination step takes $O(nr + r^{5/2})$ time, and Step 3 of ConvDecomp, which eliminates r notches, takes $O(nr^2 + r^{7/2})$ time. Combining the complexities of Step 2 and Step 3, we obtain an $O(nr^2 + r^{7/2})$ time complexity for convex decomposition of a manifold polyhedron. The space complexity of $O(nr + r^{5/2})$ follows from Lemma 3.3. □

THEOREM 3.2. *A nonmanifold polyhedron S, possibly with holes and shells and having r notches and n edges, can be decomposed into $O(r^2)$ convex polyhedra in $O(nr^2 + r^3 \log r)$ time and $O(nr + r^{5/2})$ space.*

Proof. Removal of all special notches from S is carried out in $O(n + r \log r)$ time and in $O(n)$ space, as discussed before. Let $S_1, S_2, \cdots, S_l$ be the manifold polyhedra created by this process. Let S_i have n_i edges of which r_i are reflex. Using Theorem 3.1 on each of them, we conclude that S can be decomposed into $O(r^2)$ convex polyhedra in $O(\sum_{i=1}^{l} n_i r_i^2 + r_i^{7/2}) = O(nr^2 + r^{7/2})$ time and in $O(\sum_{i=1}^{l} n_i r_i + r_i^{5/2}) = O(nr + r^{5/2})$ space. □

4. Convex decomposition under finite precision arithmetic. When implementing geometric operations stemming from practical applications, one cannot ignore the degenerate geometric configurations that often arise, as well as the need to make specific topological decisions based on imprecise finite precision numerical computations [19], [27]. We model the inexact arithmetic computations by ε-arithmetic [15], [17] where the arithmetic operations $+, -, \div, \times$ are performed with relative error of at most ε. Under this model, the absolute error in the distance computations of one polyhedral feature from another is bounded by a certain quantity $\delta = k\varepsilon B$, where B is the maximum value of any coordinate and k is a constant; see, e.g., [23]. When making decisions about the incidences of these polyhedral features (vertices, edges, facets) on the basis of the computed distances (with signs), one can rely on the sign of the computations only if the distances are greater than δ. On the other hand, if the computed distances are less than δ, one also needs to consider the topological constraints of the geometric configuration to decide on a reliable choice. In particular, in regions of uncertainty, i.e., within the δ-ball, the choices are all equally likely that the computed quantity is negative, zero, or positive. Such decision points of uncertainity where several choices exist are either "independent" or "dependent." At the independent decision points, any choice may be made from the finite set of local topological possibilities while the choices at the dependent decision points should ensure that they do not contradict any previous topological decisions. The algorithm that follows this paradigm would never fail, though it may not always compute a valid output. Such algorithms have been termed *parsimonious* by Fortune [15].

An algorithm under ε-arithmetic is called robust if it computes an output which is exact for some perturbed input. It is called stable if the perturbation required is small. Recently, in [15], [16], [21], authors have given robust and stable algorithms for some important geometric problems in two dimensions. Except [18], there is no known robust algorithm for any problem in three dimensions. The difficulty arises due to the fact that the perturbations in the positions of the polyhedral features may not render a valid polyhedron embedded in $\Re^3$. In [18], Hopcroft and Kahn discuss the existence of a valid polyhedron that admits the positions of the perturbed vertices of a convex polyhedron. The case of nonconvex polyhedra is perceived to be hard and requires understanding the deep interactions between topology and perturbations of polyhedral features of nonconvex polyhedra.

Karasick [19] gives an algorithm for the problem of polyhedral intersection where

he uses geometric reasoning to avoid conflicting decisions about polyhedral features. In this paper, we extend the results in [19] and provide an algorithm for the problem of polyhedral decomposition that also uses geometric reasoning to avoid conflicting decisions. As yet we are unable to prove our algorithm to be parsimonious. We report various heuristics we have implemented in our effort to make the decomposition algorithm more reliable in the presence of numerical errors in arithmetic computations. These heuristics are useful in the sense that they give better results in practice than the other algorithms, which assume exact arithmetic. We give an estimate of the worst-case running time bound for the algorithm under the ε-arithmetic model.

Related work. The issue of robustness in geometric algorithms has recently taken on added importance because of the increasing use of geometric manipulations in computer-aided design and solid modeling [3]. Edelsbrunner and Mucke [14] and Yap [29] suggest using expensive symbolic perturbation techniques for handling geometric degeneracies. Sugihara and Iri [28] and Dobkin and Silver [11] describe an approach to achieve consistent computations in solid modeling by ensuring that computations are carried out with sufficiently higher precision than used for representing the numerical data. There are drawbacks, however, as high precision routines are needed for all primitive numerical computations, making algorithms highly machine-dependent. Furthermore, the required precision for calculations is difficult to estimate a priori for complex problems. Segal and Sequin [26] estimate various numerical tolerances, tuned to each computation, to maintain consistency. Milenkovic [23] presents techniques for computing the arrangements of a set of lines in two dimensions robustly. He introduces the concept of *pseudolines* that preserves some basic topological properties of lines and computes the arrangements in terms of these pseudolines. Karasick [19] proposes using geometric reasoning and applies it to the problem of polyhedral intersections. Sugihara [27] uses geometric reasoning to avoid redundant decisions and thereby eliminate topological inconsistencies in the construction of planar Voronoi diagrams. Guibas, Salesin, and Stolfi [17] propose a framework of computations, called ε-geometry, in which they compute an exact solution for a perturbed version of the input. So does Fortune [15], who applies it to the problem of triangulating two-dimensional point sets. For more details on robustness, see [8].

4.1. Intersection and incidence tests. In what follows, we assume the input polyhedra to be manifold. Nonmanifold polyhedra can be handled as discussed in the earlier sections. It is clear from the discussions of our algorithm in §3 that numerical computations arise in various intersections and incidence tests. We assume minimum feature criteria for the input polyhedra wherein the distance between two distinct vertices or between a vertex and an edge is at least δ. To decide whether an edge is intersected by a plane, one must decide the classification of its terminal vertices with respect to the same plane. The same classification of a vertex is used to decide the classification of all the features incident on that vertex. This, in effect, avoids conflicting decisions about the polyhedral features. The decisions about different types of intersections and incidence tests are carried out using three basic tools, namely, (i) vertex-plane classifications, (ii) facet-plane classifications, and (iii) edge-plane classifications. The order of classifications is (i) followed by (ii) followed by (iii). In what follows, we assume that the equation of any plane $P_i : a_i x + b_i y + c_i z + d_i$ is normalized, i.e., $a_i^2 + b_i^2 + c_i^2 = 1$.

Vertex-plane classification. To classify the incidence of a vertex $v_i = (x_i, y_i, z_i)$ with respect to the plane $P : ax + by + cz + d = 0$, the normalized algebraic distance of v_i from P is computed, which is given by $ax_i + by_i + cz_i + d$. The *sign* of this computation, viz., zero, negative, or positive, classifies v_i as "on" P (zero), "below" P (negative), or

"above" P (positive) where "above" is the open half-space containing the plane normal (a, b, c). The sign of the computations is accepted as correct if the above distance of v_i from P is larger than δ. Otherwise, geometric reasoning is applied as detailed below to classify the vertex v_i with respect to the plane P. In the following algorithmic version of the vertex-plane classification, the intersection between an edge e_j incident on v_i and the plane P is computed as follows. Let e_j be incident on planes P_1, P_2, where P_i : $a_i x + b_i y + c_i z + d_i = 0$. The intersection point $\mathbf{r}$ of e_j and the plane P is determined by solving the linear system $A\mathbf{r} = \mathbf{d}$ where

$$A = \begin{bmatrix} a & b & c \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{bmatrix} \quad \text{and} \quad \mathbf{d} = [-d, -d_1, -d_2,]^T .$$

The linear system is solved using Gaussian elimination with scaled partial pivoting and iterative refinement to reduce the numerical errors.

Vertex-Plane-Classif (v_i,P)
begin
Let $v_i = (x_i, y_i, z_i)$ be a vertex incident on edges $e_1 = (v_i, w_1), e_2 = (v_i, w_2), \cdots, e_k = (v_i, w_k)$.
Let $P : ax + by + cz + d = 0$.
Compute $l = ax_i + by_i + cz_i + d$.
if $|l| > \delta$ *then* (**Comment*: unambiguously decide via the sign of distance computation*)
if $l > 0$ *then*
classify v_i as "above"
else
classify v_i as "below"
endif
else
loop
(**Comment*: if the distance computation does not yield an unambiguous classification for the vertex with respect to the plane, ensure that the "above," "below" classification is consistent with all edges incident on that vertex. If such consistency cannot be ensured then the vertex is classified as "maybeon" and left for the future *facet-plane* classifications to decide its classification consistently.*)

Search for an edge e_j incident on v_i such that $r = e_j \cap P$ is at a distance greater than δ from v_i and $w_j = (x_j, y_j, z_j)$.
Get the classification of w_j if it is already computed.
Otherwise, compute $l' = ax_j + by_j + cz_j$.
if $|l'| > \delta$ then classify w_j accordingly.
if the classification of w_j is "below" or "above" *then*
if r is in between v_i and w_j *then*
classify v_i oppositely to that of w_j
else
classify v_i same as that of w_j
endif

endif
endif
endloop
if no such edge e_j is found *then*
classify v_i as "maybeon"
(**Comment:* To be classified later in the facet-plane classifications*)
endif
endif
end.

Facet-plane classification. If a facet f_i does not lie on a plane P, the points of intersection between them should necessarily be (i) collinear with the line of intersection $f_i \cap P$, and (ii) all vertices of f_i on one side of the intersection line should have the same classification with respect to the plane P. Vertices that have been temporarily classified as "maybeon" are classified in such a way that they satisfy the above two properties (i) and (ii) as closely as possible. Note that this heuristic forces the classification of "maybeon" vertices to be more consistent than the one obtained by classifying them arbitrarily. An algorithmic version of the facet-plane classification is given below.

Facet-Plane-Classif (f_i, P)
begin
case
(i) All vertices of f_i have been classified as "maybeon":
Classify f_i as "on" the plane and change the classification of all incident vertices to "on."

(ii) At least one vertex v_u of f_i has been classified as "above," or "below," but no edge of f_i has its two vertices classified with opposite signs ("below" and "above"):
if there is only one "maybeon" vertex v_i *then*
classify v_i as "on" and consider v_i as $f_i \cap P$
else
take two "maybeon" vertices v_i,v_j and
classify v_i and v_j as "on."
Let L be the line joining v_i, v_j.
Consider L as P.
loop
for each "maybeon" vertex v_k on f_i do
if v_k is at a distance greater than δ from L *then*
if v_k and v_u lie on opposite sides of L *then*
classify v_k with the opposite classification of v_u.
else
classify v_k with the classification of v_u.
endif
endif
endloop
endif
The vertices which are still not classified
classify them as "on"
(**Comment:* these vertices are within a distance of δ
from L and hence will be collinear with L by a perturbation of
at most δ. See Fig. 11.*)

(iii) There is an edge e whose two vertices have opposite sign classifications:
if there is no other such edge *then*
let L be the line joining the intersection point of e and
P to any "maybeon" vertex v_i.
classify v_i as "on."
consider L as $f_i \cap P$.
apply methods of case (ii) to classify other "maybeon" vertices.
else
let L be the line which fits in least square sense all the points
of intersections and apply the methods of case (ii) to classify
remaining "maybeon" vertices.
endif
endcase
end.

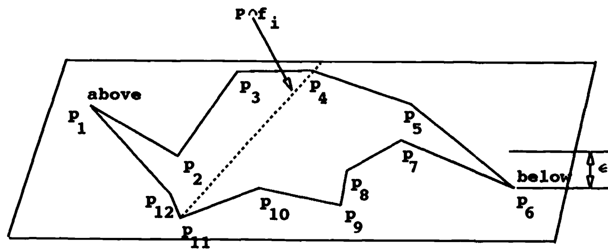


FIG. 11. *Case* (ii) *of the facet-plane classification.*

Edge-plane classification. An edge can receive any of the three classifications which are "not-intersected," "intersected," and "on". The classifications of the vertices incident on an edge e_i are used to classify it. An algorithmic version of the edge-plane classification is given below.

Edge-Plane-Classif (e_i, P)
begin
Let $e_i = (v_i, v_j)$.
case
(i) v_i and v_j are both classified as "on":
classify e_i as "on."

(ii) Only one of v_i, v_j, say, v_i is classified as "on":
classify e_i as "intersected" and consider v_i as $e_i \cap P$.

(iii) v_i and v_j are classified with one as "above" and another as "below":
classify e_i as "intersected."
compute $r = e_i \cap P$ if it has not been computed yet.
if r does not lie within e *then*
choose a point at a distance of at least δ from the vertex
which is nearest to the computed point and consider it as the intersection point of e_i and P.
endif

(iv) v_i and v_j are of same classifications and they are not "on":
classify e_i as "not-intersected."
endcase
end.

Nesting of polygons with finite precision arithmetic. The polygon nesting problem as discussed in §2.4 can be solved with finite precision arithmetic if the polygons are restricted to a class of polygons called *fleshy polygons*. A polygon P is called fleshy if there is a point inside P such that a square with the center (intersection of square's diagonals) at that point and with the sides of length $64\varepsilon B$ lies inside P. B and ε have been defined earlier.

LEMMA 4.1. *The problem of polygon nesting for k fleshy polygons with s vertices and t monotone chains can be solved in* $O(k^2+s(t+\log s))$ *time under finite precision arithmetic.*

Proof. See [4]. Since any vertical line (orthogonal to the x direction) can intersect at most t edges of a set of polygons having t monotone chains, the above time bound is obvious from the time analysis of the algorithm under finite precision arithmetic, as given in [4]. □

4.2. Description of the algorithm. The same paradigm of cutting and splitting polyhedra along the cuts is followed to produce the convex decomposition of a nonconvex, manifold polyhedron. One of the two planes supporting the facets incident on a notch is chosen as a notch plane. This ensures that no new plane other than facet-planes is introduced by the algorithm. As we have seen earlier, computations of intersection vertices involve plane equations incident on those vertices. Thus, using the original plane equations for such computations reduces the error propagation. Furthermore, this also guarantees that all input assumptions about the supporting planes of the facets remain valid throughout the iterative process of cutting and splitting the polyhedron. We apply heuristics at each numerical computation through geometric reasoning to make our algorithm as parsimonious as possible.

In the construction of GP_g, first all boundaries are computed. For this, one needs to compute the intersection vertices on the facets of S. This is carried out by the vertex-plane, edge-plane, and facet-plane classifications, as described before. Note that these classifications use heuristics that make the numerical computations more reliable. After computing all intersection vertices lying on a facet f, we sort them along the line of intersection $f \cap P_g$. Since the computed coordinates of these vertices are not exact, sorting them on the basis of their coordinates is prone to error. We use the minimum feature criteria and the orientations of the edges on a facet to obtain a topologically correct sort.

Two intersection vertices can be closer than δ if they lie on the edges meeting at a vertex. Other possibilities do not occur because of the minimum feature assumptions.

Using the orientations of these two edges on the facet f containing them, the exact ordering of the two new intersection vertices on $f \cap P_g$ can be determined. Generation of edges between intersection vertices can be carried out exactly since it does not involve any numerical computation.

The cut Q_g is selected from GP_g using the method of §3. The polygon nesting algorithm, used for this purpose, is adapted to cope with the inexact numerical computations, as stated in Lemma 4.1. The polygon nesting algorithm with inexact arithmetic computations requires all input polygons to be fleshy. Though in most of the cases this is true, we do not know how to guarantee this property throughout the decomposition process. Refinement of Q_g needs proper transferring of the edges of S that are decided to be coplanar with P_g. This is done using the following simple heuristic. For an edge e computed to be "on" the plane P_g, we check all its oriented edges incident on facets computed to be "off" the notch plane P_g. Suppose f is such a facet. Classify any vertex v of f with respect to the oriented edge of e on f. If it is on the same side of e in which f lies, e is transferred to GP_g^ℓ (GP_g^r, respectively) if v has been classified to lie in P_g^ℓ (P_g^r, respectively). It is trivial to decide the side of e in which f lies.

Splitting S about the cuts Q_g^ℓ and Q_g^r completes the cutting of S with the notch plane P_g. This step again does not involve any numerical computations.

Note that we assume the minimum feature property to be valid throughout the iterative process of cutting and splitting of polyhedra. Though for the original polyhedron it is valid, it may not be preserved throughout the entire cutting process. The method described in [26] can be used to eliminate this problem.

Complexity analysis. We use Lemmas 2.3 and 3.4 in our analysis, which is valid only under the exact arithmetic model. Nonetheless, the analysis presented here gives a good estimate of the complexity of the algorithm.

Consistent vertex-plane, edge-plane, and facet-plane classification take overall $O(p)$ time where p is the total number of edges of the polyhedron S. The above bound follows from the fact that each edge of S is visited only $O(1)$ times to determine the intersection points of S with the notch plane P_g. The sorting of intersection vertices on the facets adds $O(u \log r)$ time where u is the total number of vertices in GP_g. Once the map GP_g is constructed, it is trivial to recognize the boundary B_g containing the notch g. The methods as described in §3 can be used to determine the interesting boundaries. As discussed earlier, there are $O(t)$ interesting boundaries containing $O(t)$ monotone chains where t is the number of notches intersected by P_g. Let u' be the number of vertices on the interesting boundaries. According to Lemma 4.1, the children and parent of B_g can be determined exactly in $O(t^2 + u'(t + \log u'))$ time if the polygons corresponding to the interesting boundaries are fleshy. Detection of children and parent of the polygon containing the notch g, in effect, determines the inner and outer boundaries of Q_g. Obviously $u' = O(u)$. Combining the complexities of computing GP_g and detecting the inner and outer boundaries of Q_g, we conclude that Q_g can be computed in $O(p + t^2 + u(t + \log u) + u \log r)$ time.

At a generic instance of the algorithm, let $S_1, S_2, \cdots, S_k$ be the k distinct (nonconvex) polyhedra in the current decomposition that contain the subnotches of a notch g which is to be removed. Let p_i be the number of edges in S_i of which r_i are reflex, u_i be the number of vertices in the cross-sectional map in S_i, and t_i be the number of notches

given by

$$\begin{aligned}\Im &= O\left(\sum_{i=1}^{k}(p_i + t_i^2 + u_i(t_i + \log u_i) + u_i \log r_i)\right)\\ &= O(p + r^3 + ur + u \log u + u \log r).\end{aligned}$$

By Lemma 3.4, $u = O(n + r^2)$. This gives

$$\begin{aligned}\Im &= O(p + r^3 + (n + r^2)r + (n + r^2)\log n)\\ &= O(nr + n\log n + r^2 \log n + r^3)\\ &= O(nr + n\log n + r^3).\end{aligned}$$

To eliminate r notches, we need $O(nr^2 + nr\log n + r^4)$ time. Obviously, the space complexity is $O(p) = O(nr + r^{5/2})$. If S is a nonmanifold polyhedron, all special notches are removed from S to produce manifold polyhedra, each of which is decomposed into convex pieces by the method as discussed before. The complexity remains the same for this case. □

5. Conclusion. We have implemented our polyhedral decomposition algorithm under floating point arithmetic in Common Lisp on UNIX workstations. The numerical computations are all in C, callable from Lisp using interprocess communications. We used $\delta = 2^{-17}$ in the 32 bit machine with precision 2^{-24}. Simple examples are shown in Figs. 12 and 13.

The experimental results have been very satisfying. Test polyhedra are created, and results are displayed in the X-11 window-based SHILP solid modeling and display system [1]. The convex pieces generated can be easily triangulated to generate a triangulation of the input nonconvex polyhedra. Of course, the facet triangulations between convex pieces have to be kept consistent. To achieve this, the slicing along a notch has to be carried out through all subpolyhedra intersecting the notch plane. The time complexity does not increase for decompositions with this type of slicing, though space complexity increases to $O(nr + r^3)$. Details can be found in [9].

In finite element methods with triangular elements, nicely shaped tetrahedra are preferred to reduce ill-conditioning as well as discretization error. In [10], we have given a method to produce guaranteed quality triangulations of convex polyhedra. In Fig. 14, an example of this triangulation is shown. For clarity, we show only the triangulations on the facets. The convex decomposition method, coupled with this guaranteed quality triangulation, gives a method for guaranteed quality triangulations of nonconvex polyhedra. However, this method has the limitation that the convex polyhedra produced by the convex decomposition algorithm may be very bad in shape. An algorithm that achieves guaranteed quality triangulations of nonconvex polyhedra directly is more practical. Currently, research is going on to find such an algorithm.

intersected by the notch plane in S_i. Let $p = \sum_{i=1}^{k} p_i$, $u = \sum_{i=1}^{\kappa} u_i$, and $t = \sum_{i=1}^{\kappa} t_i$. Certainly, $k = O(r)$ and $t = O(r)$, since a notch can have at most $r - 1$ subnotches and a notch plane can intersect at most $r - 1$ notches. The time $\Im$ to remove the notch g is

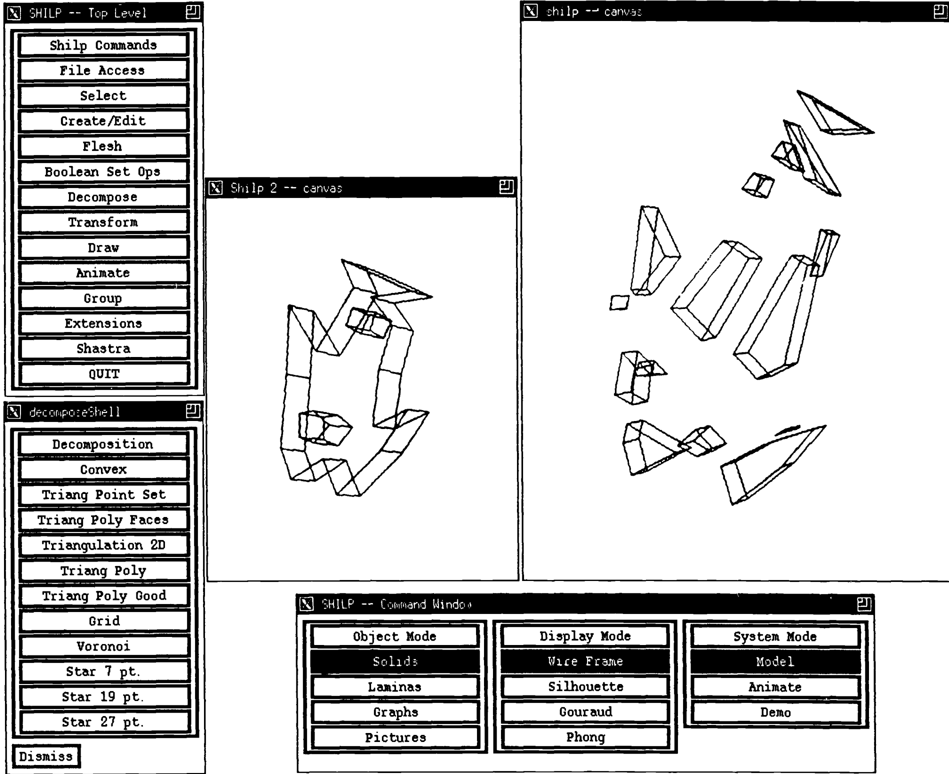


FIG. 12. *An example of convex decomposition.*

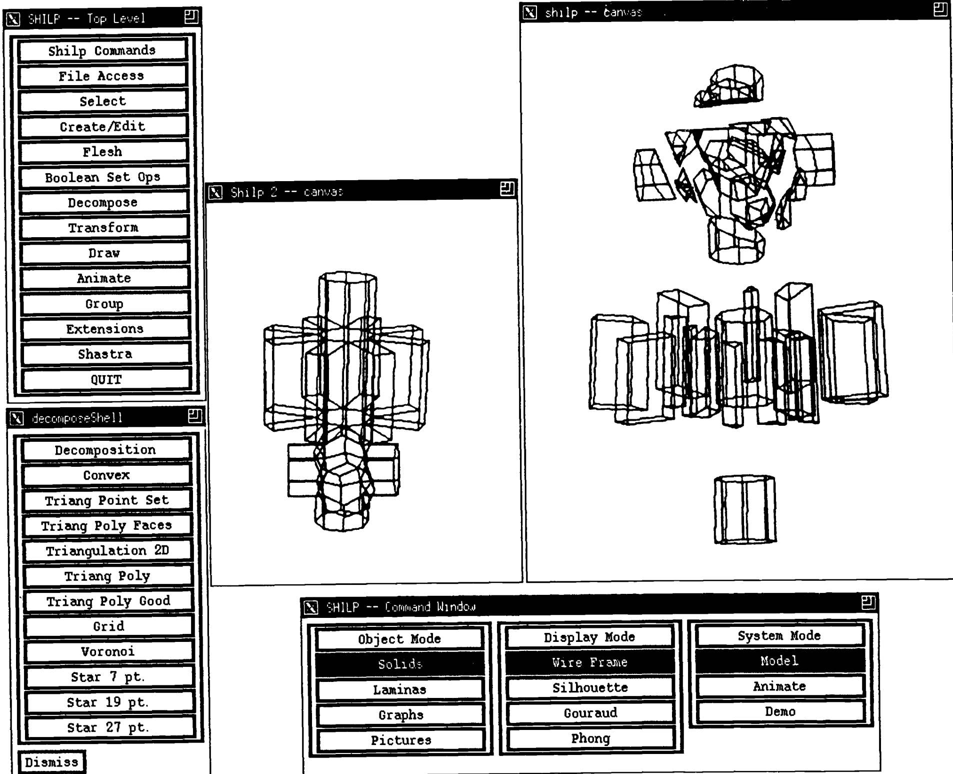


FIG. 13. *Another example of convex decomposition.*

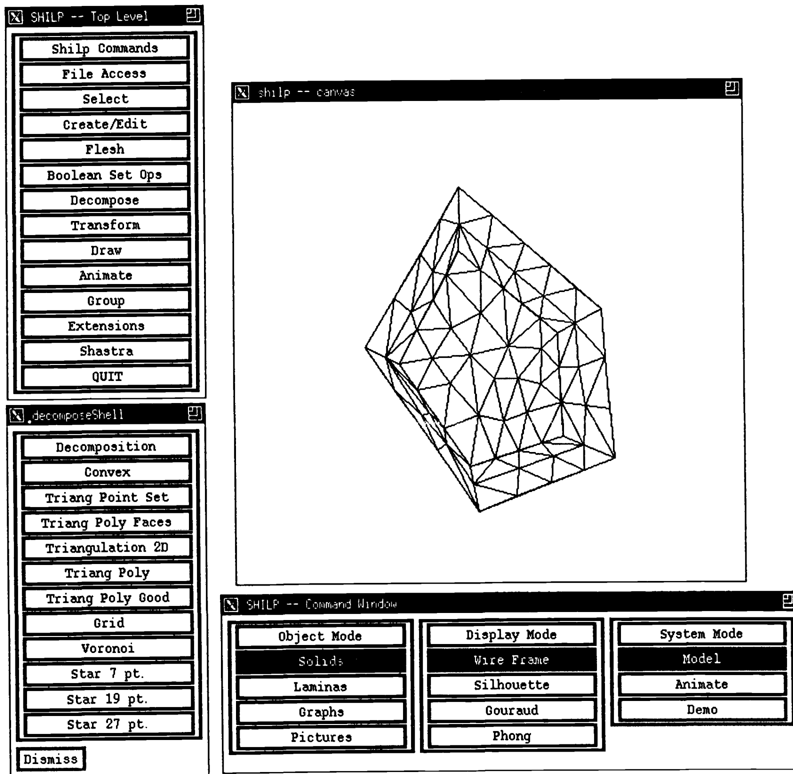


FIG. 14. *An example of a triangulation of a convex polyhedron.*

Acknowledgment. We thank Jack Snoeyink and three anonymous referees for their astute comments and suggestions.

REFERENCES

[1] V. ANUPAM, C. BAJAJ, T. DEY, I. IHM, AND S. KLINKNER, *The shilp modeling and display toolkit*, Tech. Report CSD-TR-988, Computer Science Department, Purdue University, West Lafayette, IN, 1989.

[2] M. A. ARMSTRONG, *Basic Topology*, McGraw-Hill, London, 1979.

[3] C. BAJAJ, *Geometric Modeling with Algebraic Surfaces, The Mathematics of Surfaces III*, Oxford University Press, 1988.

[4] C. BAJAJ AND T. DEY, *Polygon nesting and robustness*, Inform. Process. Lett., 35 (1990), pp. 23–32.

[5] B. CHAZELLE, *Computational geometry and convexity*, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[6] ———, *Convex partitions of polyhedra: A lower bound and worst-case optimal algorithm*, SIAM J. Comput., 13 (1984), pp. 488–507.

[7] B. CHAZELLE AND L. PALIOS, *Triangulating a non-convex polytope*, Discrete & Comput. Geom., 5 (1990), pp. 505–526.

[8] T. K. DEY, *Decompositions of polyhedra in three dimensions*, Ph.D. thesis, Department of Computer Science, Purdue University, West Lafayette, IN, 1991.

[9] ———, *Triangulation and CSG representation of polyhedra with arbitrary genus*, in Proc. 7th. ACM Symposium on Computational Geometry, 1991, pp. 364–372.

[10] T. K. DEY, C. BAJAJ, AND K. SUGIHARA, *On good triangulations in three dimensions*, in Proc. Symposium on Solid Modeling Foundations and CAD/CAM Applications (ACM/SIGGRAPH), Austin, TX, 1991, pp. 431–441.

[11] D. DOBKIN AND D. SILVER, *Recipes for geometry and numerical analysis*, in Fourth ACM Symposium on Computational Geometry, Urbana, IL, 1988, pp. 93–105.

[12] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, Heidelberg, 1987.

[13] H. EDELSBRUNNER, L. GUIBAS, J. PACH, R. POLLACK, R. SEIDEL, AND M. SHARIR, *Arrangements of arcs in the plane: Topology, combinatorics, and algorithms*, in 15th Internat. Colloquium on Automata, Languages and Programming (EATCS), 1988, pp. 214–229.

[14] H. EDELSBRUNNER AND P. MUCKE, *Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms*, in Fourth ACM Symposium on Computational Geometry, Urbana, IL, 1988, pp. 118–133.

[15] S. FORTUNE, *Stable maintenance of point-set triangulations in two dimensions*, in Proc. 30th IEEE Symposium on the Foundations of Computer Science, 1989, pp. 494–499.

[16] S. FORTUNE AND V. MILENKOVIC, *Numerical stability of algorithms for line arrangements*, in Seventh ACM Symposium on Computational Geometry, North Conway, NH, 1991, pp. 93–105.

[17] L. GUIBAS, D. SALESIN, AND J. STOLFI, *Building robust algorithms from imprecise computations*, in Fifth ACM Symposium on Computational Geometry, Saarbruchen, West Germany, 1989, pp. 208–217.

[18] J. HOPCROFT AND P. KAHN, *A paradigm for robust geometric algorithms*, Tech. Report TR 89-1044, Computer Science Department, Cornell University, Ithaca, NY, 1989.

[19] M. KARASICK, *On the representation and manipulation of rigid solids*, Ph.D. thesis, Computer Science Department, McGill University, Montréal, Quèbec, Canada, 1988.

[20] T. KOVARI, V. T. SOS, AND P. TURAN, *On a problem of K. Zarankiewicz*, Colloq. Math., 3 (1954), pp. 50–57.

[21] Z. LI AND V. MILENKOVIC, *Constructing strongly convex hulls using exact or rounded arithmetic*, in Sixth ACM Symposium on Computational Geometry, Berkeley, CA, 1990, pp. 235–242.

[22] A. LINGAS, *The power of non-rectilinear holes*, in 9th Internat. Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, Springer-Verlag, 1982, pp. 369–383.

[23] V. MILENKOVIC, *Verifiable implementations of geometric algorithms using finite precision arithmetic*, Ph.D. thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[24] J. O'ROURKE AND K. SUPOWIT, *Some NP-hard polygon decomposition problems*, IEEE Trans. Inform. Theory, 29 (1983), pp. 181–190.

[25] J. RUPERT AND R. SEIDEL, *On the difficulty of tetrahedralizing three-dimensional nonconvex polyhedra*, in Fifth ACM Annual Symposium on Computational Geometry, 1989, pp. 380–392.

[26] M. SEGAL AND C. SEQUIN, *Consistent calculations for solid modeling*, in First ACM Symposium on Computational Geometry, Baltimore, MD, 1985, pp. 29–38.

[27] K. SUGIHARA AND M. IRI, *Construction of the Voronoi diagram for one million generators in single precision arithmetic*, in First Canadian Conference on Computational Geometry, Montreal, Quebec, Canada, 1989.

[28] ———, *A solid modeling system free from topological inconsistency*, Research Memorandum RMI 89-3, Department of Mathematical Engineering and Instrumentation Physics, Tokyo University, Tokyo, Japan, 1989.

[29] C. YAP, *A geometric consistency theorem for a symbolic perturbation theorem*, in Fourth ACM Symposium on Computational Geometry, Urbana, IL, 1988, pp. 134–142.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 365–380, April 1992
© 1992 Society for Industrial and Applied Mathematics
011

FAST GOSSIPING FOR THE HYPERCUBE*

DAVID W. KRUMME†

Abstract. The gossip problem involves communicating a unique item from every node in a graph to every other node. The minimum time required to do this for the binary hypercube under two models of communication is studied. In the first model, all communication links may be used concurrently but each may only carry information in one direction at a time. In the second, weaker model each node may be involved in only one communication at a time either as sender or receiver. In both cases, simple algorithms exist that are close to optimal. This paper shows that neither of these algorithms is optimal by exhibiting faster algorithms. In the first case, an optimal algorithm is obtained.

Key words. gossiping, broadcasting, hypercube

AMS(MOS) subject classifications. 68Q20, 68R10

1. Introduction.

Gossiping generally refers to the process of distributed information dissemination and can easily be described in graph-theoretic terms. Each node in a graph initially contains a unique piece of information to be communicated to all other nodes. At each time step, a node can only communicate with those nodes that share an edge with it. Information can be combined between communications. Variants of the gossip problem involve the minimal total number of communications and the minimal total time required. Different models of communication have been proposed. Known results about gossiping are summarized in a 1988 survey paper by Hedetniemi, Hedetniemi, and Liestman [2].

In this paper, we study the minimum time required to gossip under two models of parallel communication for the binary hypercube graph. Our interest in this problem is motivated by the commercially successful hypercube multiprocessors and by the fact that gossiping commonly arises in parallel algorithms for these machines. In another paper [3] we give a more lengthy introduction to our work on this problem and derive lower bounds and upper bounds for the time to gossip in complete graphs, regular and toroidal grids, and rings. This paper is restricted to the question of upper bounds for the hypercube.

We use two models of communication. In the *simultaneous* model of communication, each node can participate in an unlimited number of communication activities at each time step. In the *pairwise* model of communication, each node can participate in just one communication event at each time step. In both models, communication is uni-directional: a communication event between two nodes consists of taking (a copy of) the tokens existing at the sending node and combining them with those already present at the receiving node. Thus the sender is unaffected while the receiver's new token set is the union of its and the sender's previous token sets.

A trivial lower bound for the time required to gossip is the diameter of the graph, which is d in the case of a hypercube of dimension d. Under the simultaneous model, it is not difficult to find $(d+1)$-step solutions and it is difficult to do better. However, this paper shows that for all $d \geq 3$ there are d-step solutions, and that for all $d \geq 4$, optimal solutions exist that are time-invariant. Time-invariant optimal solutions have

* Received by the editors March 16, 1988; accepted for publication (in revised form) February 20, 1991. This research was partially supported by Office of Naval Research contract N00014-87-K-0182 and National Science Foundation grant DCR-8619103.

† Department of Computer Science, Tufts University, Medford, Massachusetts 02155.

the interesting graph-theoretic property of assigning a direction to each edge so that the maximum directed distance between any two points is the same as the maximum distance between any two points in the undirected graph (i.e., the diameter).

Under the pairwise model, it is amazingly easy to find $2d$-step solutions and amazingly hard to do better. For example, if one uses only "parallel" transmissions, or those that move in the same direction through the same dimension at each step, then *any* sequence of $2d$ distinct such steps will solve the problem and *no* shorter sequence of such steps will solve it. Bagchi, Hakimi, Mitchem, and Schmeichel have conjectured that $2d$ steps are required [1]. However, this paper shows that the problem can be solved in $\approx 1.88d$ steps with very complicated algorithms for the 9-cube and the 17-cube. This leaves a gap between the best known lower bound of $1.44 \lg N$ [3] and $1.85 \lg N$ which is the best performance that is potentially obtainable using the approach in this paper.

Section 2 establishes notational conventions. Section 3 develops optimal solutions under the half-duplex simultaneous model. Section 4 presents a solution that is faster than $2d$ steps under the half-duplex pairwise model. Section 5 contains a concluding discussion.

2. Notation. The hypercube is bipartite and hence can be two-colored; in order to take advantage of that property, we define the *parity* $P(n)$ of a node n as $(-1)^q$, where q is the number of 1's in the binary representation of n. (The use of $+1$ for even parity and -1 for odd parity turns out to be convenient.)

We can view the $(D + d)$-dimensional hypercube as the Cartesian product of a D-cube and a d-cube. We refer to the node with coordinates (x, n) as **location** $L_{x,n}$, where $0 \le x < 2^D$ and $0 \le n < 2^d$, and we refer to the token originating at that node as $T_{x,n}$. By holding the first coordinate fixed we define subcubes $\alpha_x = \{L_{x,n} \mid 0 \le n < 2^d\}$, and by holding the second coordinate fixed we define subcubes $\beta_n = \{L_{x,n} \mid 0 \le x < 2^D\}$.

By a **strategy** we mean a rule that at some or all time steps specifies a set of hypercube edges and a direction for each of them. Note that we are limiting ourselves at the outset to half-duplex communication. We will generally avoid using elaborate notations for strategies since they are easy to describe directly. A strategy describes where transmissions occur and what direction each transmission takes, and its effect can be described in this way: let S_0 be the set of tokens present at a location L at time t, and let $S_1, S_2, \cdots, S_s$ be the sets of tokens present at neighboring nodes whose edges to L have been assigned a direction oriented toward L; then at time $t+1$ the set of tokens present at L is $\bigcup_{i=0}^{s} S_i$. We are interested only in finite sequences of steps starting at $t = 0$. Under the simultaneous communication model a strategy may use all edges at each step; under the pairwise model it may not assign directions to any two edges that are incident to the same node on the same step.

We shall call a strategy **constant** if it is the same at each time step and **alternating** if its transmissions all reverse at each step. If some transmissions are the same at each step while others reverse, we will call it **partially alternating**. In the illustrations, alternating transmissions will be depicted by arrows with hollow circles at the tails.

We will use the following notation in §3.1. Given some cube or subcube that includes a location L and some strategy Z, let $F_Z^k(L,t)$ be the set of all locations that under strategy Z receive at time $t+k$, or sooner, the tokens that are present at node L at time t. $F_Z^k(L,t)$ is the set of nodes for which L *finishes* in k steps starting at t. Similarly, let $E_Z^k(L,t)$ be the set of all nodes that under strategy Z receive at time

$t+k$, but not sooner, the tokens that are present at node L at time t. $E_Z^k(L,t)$ is the set of nodes for which L *exactly finishes* in k steps starting at t.

To describe strategies in §3.2 we shall use the following notation. If A and B describe strategies on d-cubes, define strategies $A \oplus B$, $A \otimes B$, and $A \ominus B$ on the $(d+1)$-cube formed by renumbering the vertices of B as $2^d, 2^d+1, \cdots, 2^{d+1}-1$, using the given transmission patterns within A and B, and using the following transmissions between pairs of neighboring vertices in A and B: (1) for $A \otimes B$, use transmissions that go from vertices in A to their neighbors in B at each time step; (2) for $A \oplus B$, use transmissions that go from all even-parity vertices in A to their neighbors in B and transmissions from B to A for all other vertices in A; (3) for $A \ominus B$, use transmissions as in $A \otimes B$ for the first time step, but then reverse their direction at each time step after that. Given a strategy A, we can obtain a strategy $\overline{A}$ by reversing the directions of all transmissions at all times, $\widehat{A}$ by reversing all directions at the second, fourth, etc. time step, and $\widetilde{A}$ by making all transmissions occur one step sooner (and deleting the first step). Now define $A\times \equiv A \otimes \overline{A}$ $A+ \equiv A \oplus \overline{A}$, and $A- \equiv A \ominus \widetilde{A}$.

In working with the binary representations of numbers in §4, we number the bits from 0 starting at the right so that bit i has value 2^i. The notation $b_i(n)$ selects bit i of n and is 0 or 1. For convenience, we define $b_{-1}(n)$ as 1. The notation $\tilde{b}_i(n)$ $(i \geq 0)$ represents the value resulting from clearing bit i of n. Thus $\tilde{b}_i(n) = n - 2^i b_i(n)$. We extend the notation so that $\tilde{b}_{i_1,i_2,\cdots,i_p}(n) = \tilde{b}_{i_1}(\tilde{b}_{i_2}(\cdots(\tilde{b}_{i_p}(n))\cdots))$ represents clearing bits $i_1, i_2, \cdots, i_p$. Observe that n and m differ only in bit i if $b_i(n) \neq b_i(m)$ and $\tilde{b}_i(n) = \tilde{b}_i(m)$. We define a *k-neighborhood* of n by $N_k(n) = \{m \mid \tilde{b}_{0,1,2,\cdots,k-1}(m) = \tilde{b}_{0,1,2,\cdots,k-1}(n)\}$. $N_k(n)$ is the k-dimensional set of points that do not differ from n in any bits $j \geq k$.

3. Simultaneous communication. We will construct a family of solutions to the token exchange problem that are optimal under the assumptions of the half-duplex simultaneous model of communication. No solution can take fewer steps than the diameter of the cube, and for diameters greater than 2 we will find solutions that take exactly that many steps. Before proceeding the reader may find it amusing to attempt to discover a three-step solution for the 3-cube.

It is not difficult to find *near-optimal* solutions to this problem that take just one step more than the diameter. Our construction of optimal solutions uses such $(d+1)$-step solutions on subcubes α_x and $(D+1)$-step solutions on subcubes β_n. It is immediate that such an approach can solve the full problem in $D+d+2$ steps, but with care the combined strategy can be made to solve the problem in $D+d$ steps, which is optimal.

3.1. A general construction. In this section we give a general method for constructing $(D+d)$-step solutions to the token exchange problem for the hypercube under the half-duplex simultaneous communication model. The method is based on the fact that there are several paths between any pair of nodes, and if a strategy is slow along one path, we will arrange that there is some other path along which the strategy is faster. The method uses two pairs of strategies: A_1 and A_2 on the d-dimensional subcubes α_x and B_1, and B_2 on the D-dimensional subcubes β_n. We will require that the strategies satisfy certain properties and that they be combined so as to satisfy certain constraints, and then we prove that the resultant overall strategy gives a $(D+d)$-step solution to the problem. Then we review a list of possible choices for the two pairs of strategies. Although the general construction has very generally stated time dependencies, all of our actual strategies are either the same at every time

step or alternate with some direction-reversals at every step.

All four strategies must be near-optimal at all times: A_1 and A_2 must solve the d-cube in $d+1$ steps when started at any time t, and B_1 and B_2 must solve the D-cube in $D+1$ steps when started at any t. A_1 and A_2 must satisfy the following property for all L, $t \geq 0$, and $\{i,j\} = \{1,2\}$: (1) $E^{d+1}_{A_i}(L,t) \subseteq F^d_{A_j}(L,t+D)$. Roughly, this says that if L's token goes to M maximally slowly in one of the strategies, then it is one step faster in the other if started after a delay of D. Strategies B_1 and B_2 must satisfy the following property for all L, $t_1 \geq 0$, $t_2 \geq 0$, and $\{i,j\} = \{1,2\}$: (2) $E^{D+1}_{B_i}(L,t_1) \subseteq F^{D-1}_{B_j}(L,t_2)$. Property (2) says that nodes whose tokens finish maximally slowly in one strategy are two steps faster in the other. Note that property (2) implies property (1), so that any strategies suitable for B_1 and B_2 are also suitable for A_1 and A_2.

Now in the combined strategy we must have two properties, (3) and (4). (3) If for some n, x, y, i, and t, $L_{y,n} \in E^D_{B_i}(L_{x,n},t)$, then A_1 is used on α_x if and only if A_2 is used on α_y. This condition says that subcube pairs on which both B_1 and B_2 are one step short of maximally slow use opposite strategies. (4) Let x be arbitrary and suppose $L = L_{x,n}$ and $M = L_{x,m}$ are such that for some t_1 we have $M \notin F^{d-1}_{A_1}(L,t_1)$ and for some t_2 we have $M \notin F^{d-1}_{A_2}(L,t_2)$; then we must have that B_1 is used on β_n if and only if B_2 is used on β_m. This condition says that node pairs for which both A_1 and A_2 can be slow must use opposite strategies. (Note that since the property uses A_1 and A_2 identically, it is independent of x; x appears only because the notation requires its presence.)

THEOREM 3.1. *Any combined strategy, as described above, satisfying properties* (1), (2), (3), *and* (4) *solves the* $(D+d)$*-dimensional token exchange problem in* $D+d$ *steps.*

Proof. Let $L_{x,n}$ be an arbitrary location; we will follow the strategies starting at $t=0$ and show that token $T_{x,n}$ reaches all other locations by time $t = D+d$. Let y refer to any other subcube. We distinguish three cases according to whether $T_{x,n}$ reaches $L_{y,n}$ in fewer than D, exactly D, or $D+1$ steps. (I) If the strategy (B_1 or B_2) used on β_n carries $L_{x,n}$ to $L_{y,n}$ in $D-1$ or fewer steps, then since either A_1 or A_2 will carry the tokens from $L_{y,n}$ to all other sites on α_y in $d+1$ or fewer steps, all sites on α_y are reached in $D+d$ steps at most. See Fig. 1. (II) If the strategy used on β_n carries

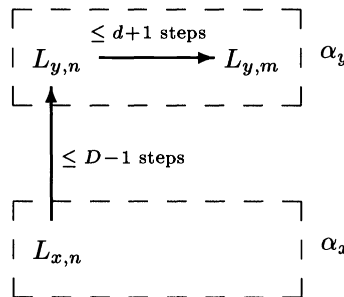


FIG. 1. *Case* (I).

$L_{x,n}$ to $L_{y,n}$ in D steps, let A_j be the strategy used on α_y and let $\{i,j\} = \{1,2\}$. See Fig. 2. All locations $L_{y,m} \in F^d_{A_j}(L_{y,n},D)$ are completed in $D+d$ steps, so we only need to be concerned about those for which $L_{y,m} \in E^{d+1}_{A_j}(L_{y,n},D)$. In this case we reason as follows. Property (3) ensures that A_i is used on α_x. We claim that

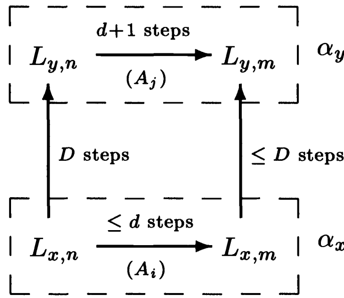


FIG. 2. *Case* (II).

$L_{x,m} \in F^d_{A_i}(L_{x,n}, 0)$; this is because otherwise we would have $L_{x,m} \in E^{d+1}_{A_i}(L_{x,n}, 0)$, which according to property (1) would mean that $L_{y,m} \in F^d_{A_j}(L_{y,n}, D)$, whereas we have assumed that $L_{y,m} \in E^{d+1}_{A_j}(L_{y,n}, D)$. Now we simply observe that $T_{x,n}$ reaches $L_{x,m}$ in d or fewer steps and we can guarantee that it then reaches $L_{y,m}$ in D more steps: if the same strategy is used on β_m as on β_n, then this is true by the assumption under (II); otherwise if the opposite strategy is used, it must also take D or fewer steps, because if it took $D + 1$ steps then property (2) would be violated. (III) If the strategy $B_p$used on β_n carries $L_{x,n}$ to $L_{y,n}$ in $D+1$ steps, let A_j be the strategy used on α_y. Now if $L_{y,m} \in F^{d-1}_{A_j}(L_{y,n}, D + 1)$, it will receive the token by time $D + d$. Otherwise we check which strategy is used on β_m. If it is B_q with $q \neq p$, then by property (2) it carries tokens from $L_{x,m}$ to $L_{y,m}$ in $D - 1$ or fewer steps, and since going from $L_{x,n}$ to $L_{x,m}$ initially can take at most $d + 1$ steps, the combined total would be no greater than $D + d$. See Fig. 3. This leaves us with the case where B_p

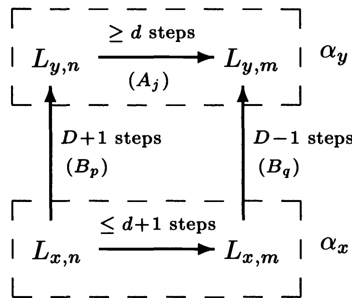


FIG. 3. *Case* (III), B_q *used on* β_m.

is used on β_m with the result that it takes $D + 1$ steps to go from $L_{x,m}$ to $L_{y,m}$. Let z be such that a token at $L_{z,m}$ at time $D + d - 1$ is carried to $L_{y,m}$ at time $D + d$. (There must be one such z, since by assumption B_p takes $D + 1$ steps at any t.) See Fig. 4. Since $L_{z,m} \in E^D_{B_p}(L_{x,m}, d - 1)$, we can invoke property (3) and say that A_i is used on α_x and A_h is used on α_z where $\{i, h\} = \{1, 2\}$. Now we claim that either (a) $L_{x,m} \in F^{d-1}_{A_i}(L_{x,n}, 0)$ or (b) $L_{z,m} \in F^{d-1}_{A_h}(L_{z,n}, D)$. This claim follows directly from property (4) since we have already assumed that B_p is used on both β_n and β_m. Now in case (a) it can be seen that $T_{x,n}$ reaches $L_{x,m}$ under A_i at time $d - 1$ and from there proceeds via B_p to $L_{z,m}$ in D more steps. In case (b) we observe that $T_{x,n}$ reaches $L_{z,n}$ at time D using B_p, and then proceeds to $L_{z,m}$ via A_h in $d - 1$ more steps. Thus in either case $T_{x,n}$ reaches $L_{z,m}$ in $D + d - 1$ steps, and then in one final

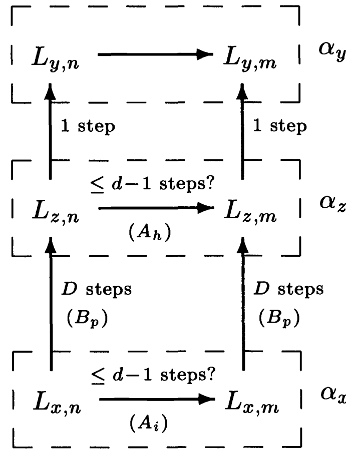


FIG. 4. *Case* (III), B_p *used on* β_m.

step reaches $L_{y,m}$ at time $D + d$. □

The following section gives a variety of choices for the A_i and B_p, yielding a large number of different optimal strategies, for cubes as small as dimension 3.

3.2. Optimal strategies. This section presents individual strategies that can be used in the general construction to produce combined strategies that are optimal under the simultaneous model.

Define two basic constant strategies F and R on the 1-cube: for F ("forward"), node 0 transmits to node 1 at each time step; for R ("reverse"), node 1 transmits to node 0 at each time step. Figure 5 shows some important basic strategies we can define using the notation introduced in §2. $R+$ is a 4-node circular pattern. $\widehat{F}$, $\widehat{F+}$, $\widehat{F++}$, etc. are alternating strategies on the 1-cube, 2-cube, 3-cube, etc. that can be described in this way: two-color the cube red and black and first have all red nodes transmit to all their neighbors, then have all black nodes transmit to all their neighbors, then repeat. The last pattern in the illustration is an important one whose structure is best shown in the formula $\overline{\overline{\overline{F}}}+++$ but that can be represented more succinctly by the equivalent formula $\overline{R+++}$.

Now we are in a position to describe strategies usable for (A_1, A_2) and (B_1, B_2) in the above construction. For (B_1, B_2), $(R+, \overline{R+})$ is the only pair known to be usable, because they are the only pair known for which properties (3) and (4) can be satisfied. Although other strategies have been found that satisfy property (2) such as $\overline{R+++}$ and larger analogues, considerable effort has been expended without success to find such strategies that can also be arranged as specified in (3) and (4). The major problem is (3): it requires that if *either* B_1 or B_2 takes D steps to go from L to M, then opposite A strategies must be used at L and M. The way to look at this is that the construction induces a two-coloring so that if B_1 or B_2 takes D steps between L and M, then L and M have opposite colors; this turns out to be a hard condition to satisfy. For the $R+$ and $\overline{R+}$ strategies, the node pairs that take $D = 2$ steps in either strategy are at distance 2 apart in the 2-cube, so the coloring is achieved by dividing the 2-cube in half and using one color throughout each half.

Strategies for A_1 and A_2 are relatively easy to find. For example, one can use any example from Fig. 5 as A_1 and its reverse $\overline{A_1}$ as A_2. Given constant, alternating, or partially alternating (A_1, A_2), then (A_1-, A_2-) is also usable. An illustration of $\widehat{F}-$

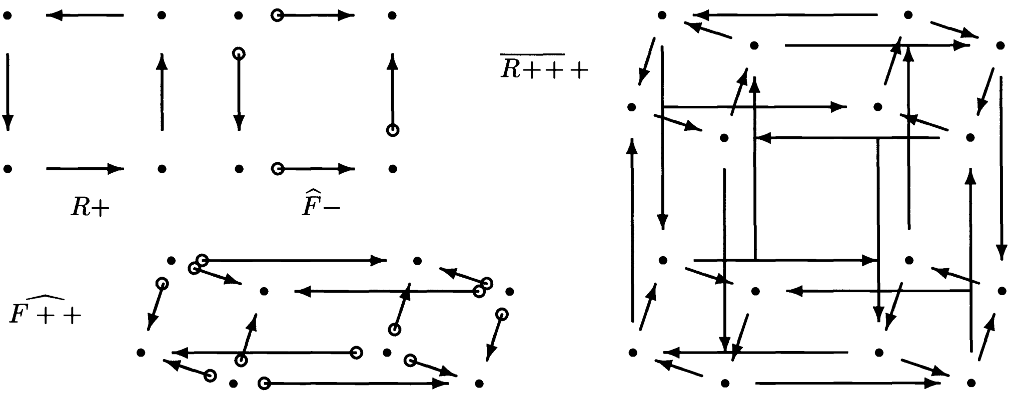


FIG. 5. *Basic strategies. Arrows with circles on them alternate directions at each time step.*

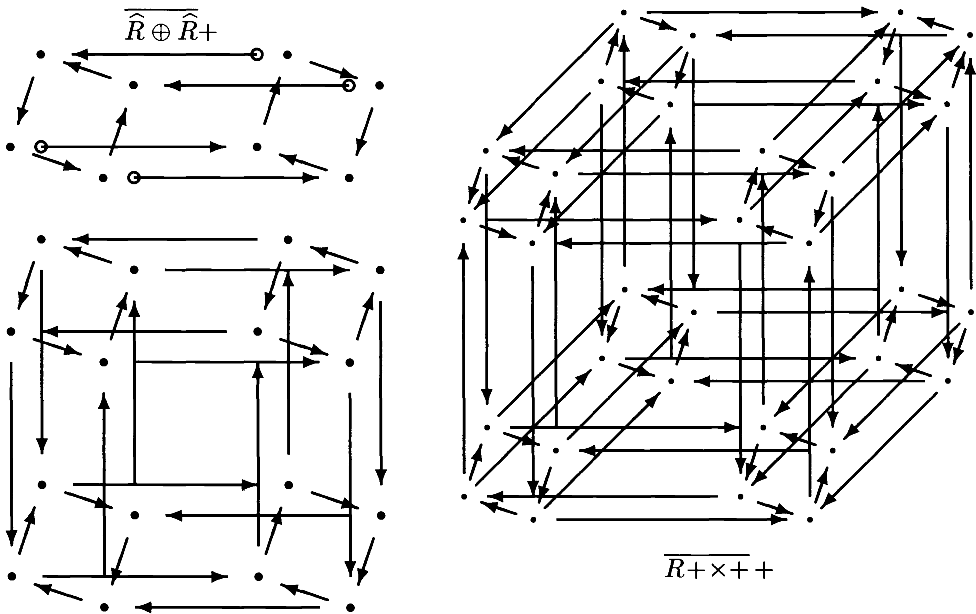


FIG. 6. *Some optimal strategies. There is no notation for the one at the lower left.*

is shown in Fig. 5.

Let (S_1, S_1') be constant strategies for a d_1-cube satisfying (1) and let (S_2, S_2') be constant strategies for a d_2-cube satisfying (2), such as $(R+, \overline{R+})$ or $(\overline{R+++}, \overline{R+++})$. Form the Cartesian product of a d_2-cube and a d_1-cube and create strategies A_1 (A_2) for it as follows. On each subcube defined by holding the second coordinate fixed, use S_2 (S_2'); on each subcube defined by holding the first coordinate fixed, use S_1 (S_1') if

the first coordinate has even parity and use S_1' (S_1) if it has odd parity. $\overline{R\text{+++}}$ and its reverse can be obtained in this way by using $R+$ as S_1, and S_2 and $\overline{R+}$ as S_1' and S_2'. In general, (A_1, A_2) obtained in this way satisfy not only (1) but also (2), and thus they can be reused as (S_2, S_2') in a repeated application of this construction.

Optimal solutions for the hypercube token exchange problem under the simultaneous model are now obtained by using any of the above methods to generate A_1 and A_2 strategies for a d-cube and using $R+$ and $\overline{R+}$ for B_1 and B_2 with $D = 2$. A_1 is used on subcubes α_0 and α_1 and A_2 is used on α_2 and α_3. The choices of whether to use B_1 or B_2 on each β_n is generally determined by property (4). The smallest case where this works uses $\widehat{F}$ and $\widehat{R}$ with $d = 1$, yielding a solution for the 3-cube. For all larger values of d, several different optimal strategies are obtained. Figure 6 shows the solution for the 3-cube, the constant solution for the 4-cube obtained by using $R+$ and $\overline{R+}$ for A_1 and A_2, and the constant optimal strategy for the 5-cube that is described in the next section.

3.3. Constant optimal strategies. Constant strategies have desirable local properties: at each node a constant strategy can be controlled with small table without retaining any state information, and without regard to the states of neighbors. Constant optimal strategies have the following graph-theoretic interpretation. Define the *directed diameter* of a directed graph as the maximum length, over all (v_1, v_2), of a shortest path from v_1 to v_2. Now, given an undirected graph, consider the problem of choosing a direction for each edge so that the resulting directed graph has directed diameter equal to the diameter of the original graph. This is the problem of finding constant optimal strategies.[1]

Constant optimal strategies for the hypercube can be based on Theorem 3.1 by using $(R+, \overline{R+})$ for (B_1, B_2) and various constant strategies for (A_1, A_2). For even values of d there are many choices for (A_1, A_2) beginning with $(R+, \overline{R+})$, which yields a constant optimal strategy for the 4-cube.

For odd values of d, the strategies are harder to find. However, constant optimal strategies for all odd $d \geq 5$ can be based on either $\overline{R+\times++}$ (see Fig. 6) or $\overline{R\times+++}$, which are both constant optimal strategies for the 5-cube. Since an optimal strategy automatically satisfies property (1), either of these is usable as A_1 and A_2 in the construction for the 7-cube, and similar constructions can be carried forward to all higher odd values of d.

It is not difficult to show by exhaustive search that there are no constant strategies satisfying property (1) for $d = 3$, which means that there are no constant optimal strategies for the 3-cube and furthermore that such strategies for the 5-cube cannot be based on Theorem 3.1. In contrast to $\overline{R+\times++}$ and $\overline{R\times+++}$, which complete in five steps on the 5-cube, the related constant strategies $\overline{R++\times+}$ and $\overline{R++\;++}$ require seven steps on the 5-cube and hence are not even usable as A_1 and A_2 in a solution for the 7-cube.

3.4. Summary. Optimal solutions to the token exchange problem under the half-duplex simultaneous model of communication are possible for all hypercubes of dimension 3 or higher. For dimension 4 and higher, optimal strategies that are time-invariant exist. As the size of the hypercube increases, strategies are easier to find and an increasing variety is available.

---

[1] **Added in proof:** See also [4] for constant optimal strategies for all $d \geq 4$ obtained by repeated applications of $\oplus$ to the pattern shown in the lower left of Fig. 6.

On some hardware, such as the NCUBE, it may be more efficient for a node to *broadcast* its tokens to its neighbors than to send to some and receive from others at one step. Thus the nonoptimal alternating strategy $F\widehat{+\cdots+}$ might be the best choice since it is close to optimal and can be done using local broadcasts exclusively. (One cannot do better using only local broadcasts, since a node that does not send its token initially cannot transmit it to the node at distance d in fewer than d additional steps.)

Optimal strategies can be extended to larger cubes in several ways. Given optimal strategies for d_1- and d_2-cubes, an optimal strategy for the Cartesian product of the two cubes is obtained by using the two strategies in succession on the appropriate subcubes. Not only does this produce an optimal strategy for the $(d_1 + d_2)$-cube, but it uses each edge no more than half the time on average. Another approach can be based on the fact that $S-$ is optimal if S is an optimal constant, alternating, or partially alternating strategy.

4. Pairwise communication. Under the half-duplex pairwise model of communication, an appealing but false conjecture is that $2d$ steps are necessary and sufficient [1]. If at each time step the transmissions go uniformly in the same direction through the same dimension, then *any* distinct sequence of $2d$ such steps solves the problem and *no* shorter sequence does. Furthermore, in an arbitrary graph, if one uses an approach based on compression to a single node followed by expansion to all nodes, then $2r$ is a lower bound where r is the *radius* of the graph: the minimum over all points p of the maximum distance from p to any other point in the graph; the radius of the d-cube is d.

This section shows that the conjecture is false by presenting a 17-step algorithm for the 9-cube and a 32-step algorithm for the 17-cube. The algorithm uses the Cartesian product of a D-cube and a $(D+1)$-cube, or in other words, it requires that $d = D + 1$.

4.1. Definitions.

DEFINITION 4.1. A subcube x is type A_s^k, where $0 \le k \le d$ and $0 \le s \le D$, if and only if for all n such that $P(n) = (-1)^k$, $L_{x,n}$ has tokens $T_{z,m}$ for all $z \in N_s(x)$ and all $m \in N_k(n)$.

DEFINITION 4.2. A subcube is type B_s^k, where $1 \le k \le d$ and $0 \le s \le D$, if and only if it is both type A_s^k and type A_s^{k-1}.

Remark. Half the locations on a subcube are relevant to property A_s^k, while B_s^k depends on all locations. A_s^k and A_s^{k-1} are independent properties because they depend on different locations. Although it is true that if a subcube is type A_s^k (B_s^k), then it is also type A_s^{k-2} (B_s^{k-1}), we will only be interested in the maximal values of k for which one can say that a subcube is of type A_s^k (B_s^k).

DEFINITION 4.3. C_k for $1 \le k < d$ is a two-step transformation on a subcube x defined as follows. In the first step each location $L_{x,n}$ such that $P(n) = (-1)^k$ transmits to $L_{x,m}$ where m and n differ only in bit $k-1$. In the second step the same locations transmit to locations $L_{x,m}$ where m and n differ only in bit k.

Remark. We could use any bit less than or equal to $k - 1$ in the first step and still achieve our results below; $k - 1$ was chosen arbitrarily. Figure 7 shows an instance of C_k for the 3-cube. Observe that C_k is legal under the pairwise model of communication.

LEMMA 4.1. *C_k applied to a subcube of type A_s^k produces a subcube of type B_s^{k+1}.*

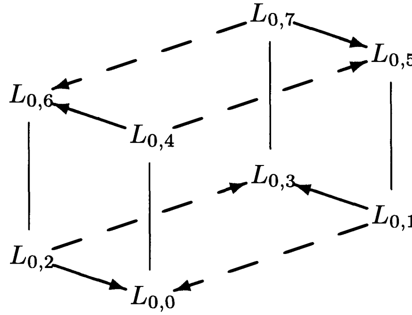


FIG. 7. C_1 *on* α_0. *Dashed arrows represent the first step and solid arrows the second.*

Proof. The locations relevant to the definition of A_s^k all transmit in both steps of C_k and therefore are unchanged, so that after C_k the subcube is still of type A_s^k. We must show that it is also A_s^{k+1}. Let $P(n) = (-1)^{k+1}$ and let p differ from n only in bit $k-1$ and q differ from n only in bit k. Then $L_{x,p}$ transmits to $L_{x,n}$ in step 1 and $L_{x,q}$ transmits to $L_{x,n}$ in step 2. Now for all $m \in N_{k+1}(n)$, either $m \in N_k(p)$ or $m \in N_k(q)$. Let $z \in N_s(x)$. If $m \in N_k(p)$, since $P(p) = (-1)^k$, $L_{x,p}$ has token $T_{z,m}$ (because of the A_s^k hypothesis) and transmits it to $L_{x,n}$ in the first step. Similarly, if $m \in N_k(q)$, $L_{x,q}$ has $T_{z,m}$ and transmits it to $L_{x,n}$ in the second step. $\square$

DEFINITION 4.4. $E_{i,1}$ and $E_{i,-1}$ are one-step transformations defined on pairs of subcubes x and y where x and y differ only in bit i. In $E_{i,1}$, location $L_{x,n}$ transmits to $L_{y,n}$ if $P(n) = (-1)^{b_i(x)}$, and otherwise $L_{y,n}$ transmits to $L_{x,n}$. $E_{i,-1}$ is the reverse (all directions reversed) of $E_{i,1}$. The definitions can be summarized by saying that for $u = \pm 1$, under $E_{i,u}$, $L_{z,n}$ transmits if and only if $P(n) = u(-1)^{b_i(z)}$.

Remark. Note that since x and y differ in bit i, the definition applied to x is consistent with its application to y, and thus $E_{i,1}$ and $E_{i,-1}$ are well defined. Figure 8 shows $E_{0,1}$ for two 2-cubes.

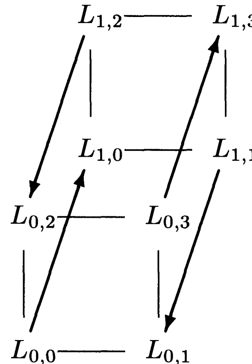


FIG. 8. $E_{0,1}$ *for* α_0 *and* α_1 $(d=2)$.

LEMMA 4.2. *Suppose* $E_{i,u}$ *is applied to two subcubes that are both of type* B_i^k *and that* x *is one of the subcubes. Then* x *becomes type* A_{i+1}^{k-1} *if* $(-1)^{b_i(x)+k} = u$, *and it becomes type* A_{i+1}^k *if* $(-1)^{b_i(x)+k} = -u$.

Proof. Let y differ from x only in bit i so that $E_{i,u}$ involves x and y. Note that x and y initially are both type A_i^k and A_i^{k-1}. (I) Suppose $(-1)^{b_i(x)+k} = u$, or equivalently, $(-1)^{k-1} = u(-1)^{b_i(x)+1} = u(-1)^{b_i(y)}$. Let n be such that $P(n) =$

$(-1)^{k-1}$ and let $m \in N_{k-1}(n)$ and $z \in N_{i+1}(x)$. If $b_i(z) = b_i(x)$, then $z \in N_i(x)$ and since x initially is type A_i^{k-1}, $L_{x,n}$ already contains $T_{z,m}$. If $b_i(z) \neq b_i(x)$, then $z \in N_i(y)$ and since y initially is type A_i^{k-1}, $L_{y,n}$ initially contains $T_{z,m}$. Now we observe that by assumption, $P(n) = (-1)^{k-1} = u(-1)^{b_i(y)}$ so that according to the definition of $E_{i,u}$, $L_{y,n}$ transmits to $L_{x,n}$. Thus in either case, $L_{x,n}$ ends up with $T_{z,m}$ and thus x is type A_{i+1}^{k-1} after $E_{i,u}$. (II) Suppose $(-1)^{b_i(x)+k} = -u$, or equivalently, $(-1)^k = -u(-1)^{b_i(x)} = u(-1)^{b_i(y)}$. Let n be such that $P(n) = (-1)^k$ and let $m \in N_k(n)$ and $z \in N_{i+1}(x)$. If $b_i(z) = b_i(x)$, then $z \in N_i(x)$ and since x initially is type A_i^k, $L_{x,n}$ already contains $T_{z,m}$. If $b_i(z) \neq b_i(x)$, then $z \in N_i(y)$ and since y initially is type A_i^k, $L_{y,n}$ initially contains $T_{z,m}$. As before, we observe that by assumption, $P(n) = (-1)^k = u(-1)^{b_i(y)}$ so that $L_{y,n}$ transmits to $L_{x,n}$. Thus in either case $L_{x,n}$ ends up with $T_{z,m}$ and thus x is type A_{i+1}^k after $E_{i,u}$. □

DEFINITION 4.5. Define $a_i(x)$ for $0 \leq i \leq D$ in this way.

$$
\begin{aligned}
a_0(x) &= 1, \\
a_1(x) &= 1 + b_0(x), \\
a_{i+1}(x) &= a_i(x) + b_i(x)b_{i-1}(x) + (1 - b_i(x))(1 - b_{i-1}(x)) \quad \text{for } i > 1.
\end{aligned}
$$

We defined $b_{-1}(x)$ to be 1, so that $a_i(x)$ is one plus the number of times $b_j(x)$ is the same as $b_{j-1}(x)$ as one proceeds through $j = 0, 1, \cdots, i-1$. Observe that $a_i(x)$ does not depend on any bits greater than or equal to i.

DEFINITION 4.6. E_i is a one-step transformation defined in this way: for every pair of subcubes x and y that differ only in bit i, apply $E_{i,u}$ where $u = (-1)^{a_i(x)+b_{i-1}(x)} = (-1)^{a_i(y)+b_{i-1}(y)}$.

Remark. Observe that E_i is well-defined and legal under the pairwise model for $0 \leq i < D$. Figure 9 illustrates E_1 for a 2-cube of 2-cubes.

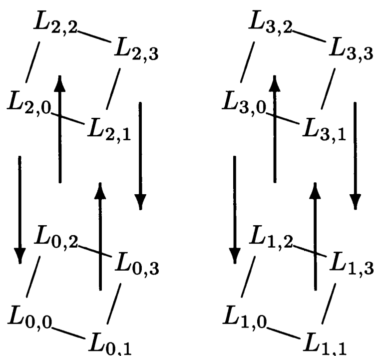


FIG. 9. E_1 $(D=d=2)$.

4.2. Overview of the algorithm. We are now ready to present the algorithm for solving the token exchange problem on the $(D+d)$-cube in fewer than $2(D+d)$ steps. Before addressing the details, let us preview it in general terms. Initially all subcubes are type A_0^0 and we work to make them A_D^d from which they can be completed in one more step. At our disposal are the special moves C_k and $E_{s,u}$, as well as ordinary "copying" moves whereby locations that have acquired complete sets of tokens simply send them to their neighbors. C_k can change two A_s^k subcubes into

B_s^{k+1} subcubes from which $E_{s,u}$ can produce one A_{s+1}^{k+1} and one A_{s+1}^{k}. If we were to follow a pattern that ignored the "lesser" result A_{s+1}^{k} and proceeded with the A_{s+1}^{k+1}, combining it with other similar ones, we could fairly easily produce an overall solution in exactly $2(D+d)$ steps, by eventually building up a few A_D^d subcubes and copying them to all subcube sites. The key to the faster solution lies in using also some of the "lesser" results to produce ultimately some additional A_D^d subcubes, so that fewer steps are required in the final copying.

A natural pattern to use is to apply C_{k_1} then E_{0,u_1} then C_{k_2} then E_{1,u_2} and so on, where the values $k_1, u_1, k_2, u_2, \cdots$ depend on the subcubes' locations. There are two difficulties. First, if $E_{s,u}$ is to be usefully applied, both subcubes involved must be of type B_s^k for some k, and thus the pattern must ensure that this always occurs. Second, when A_D^d subcubes are finally generated, they must occur in locations from which they can be propagated everywhere by copying without mutual interference; because of the way that they are generated, they tend to arise as neighbors or near-neighbors and not in good positions for efficient propagation, so care must be taken to ensure that they emerge in reasonable positions. The only difference between $E_{i,1}$ and $E_{i,-1}$ is which resultant subcube is the lesser one, and by choosing u in $E_{i,u}$ one can influence the locations where the various types of subcubes arise. The actual algorithm uses the function a_j of the previous section to define u in $E_{s,u}$ so that both of the difficulties are taken care of.

The algorithm consists of an initial step, a phase called P, a phase called Q, and a final step. The initial step creates A_0^1 subcubes everywhere. P is the heart of the algorithm, consisting of D three-step sequences $P_0, P_1, \cdots, P_{D-1}$, where each three-step sequence consists of C_k followed by $E_{s,u}$. P produces one A_D^d and a number of lesser subcubes. Q involves using C_k to complete the lesser subcubes $A_D^{d-1}, A_D^{d-2}, \cdots$ and concurrently propagating A_D^d subcubes by copying. The final step finishes the A_D^d subcubes so that all locations have all tokens. The reader may find it useful to refer to Fig. 10 while going through the following description of the algorithm.

4.3. The algorithm.

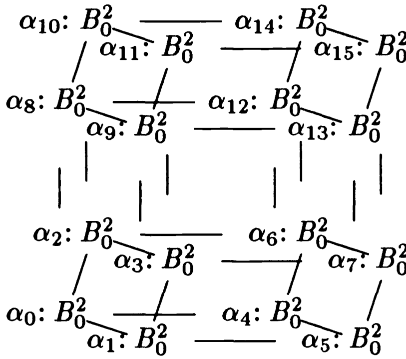
Initial step. For all x and all n such that $P(n)=1$, let m differ from n only in bit 0 and transmit from $L_{x,n}$ to $L_{x,m}$.

FACT 4.1. *After the initial step all subcubes are of type A_0^1.*

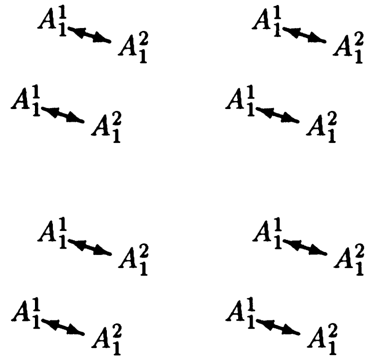
Phase P. For $0 \le i < D$ apply P_i, defined as follows, in sequence. First, for each subcube z apply the two-step sequence $C_{a_i(z)}$. Then apply E_i.

THEOREM 4.1. *After P_i each subcube α_z is of type $A_{i+1}^{a_{i+1}(z)}$.*

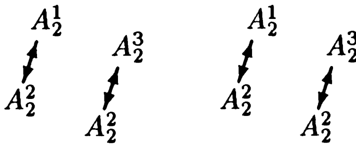
Proof. Before P_0 each subcube z is of type $A_0^1 = A_0^{a_0(z)}$. Assume then that before P_i each subcube is of type $A_i^{a_i(z)}$ and we shall show that after P_i it is $A_{i+1}^{a_i+1}(z)$. Lemma 4.1 says that after the two steps of $C_{a_i(z)}$ each cube becomes type $B_i^{a_i(z)+1}$. Now E_i involves pairs x and y where x and y differ only in bit i. This means that $a_i(x)=a_i(y)$ so that both x and y are type $B_i^{a_i(x)+1}$ and Lemma 4.2 applies. Now let z be either x or y and distinguish two cases according to whether $b_i(z)=b_{i-1}(z)$. (I) If $b_i(z)=b_{i-1}(z)$, first note that $a_{i+1}(z)=a_i(z)+1$. By Definition 4.6, $E_i = E_{i,u}$ where $u=(-1)^{a_i(z)+b_{i-1}(z)}$, so since z is type $B_i^{a_i(z)+1}$ we find in applying Lemma 4.2 that $(-1)^{b_i(z)+a_i(z)+1} = (-1)^{b_{i-1}(z)+a_i(z)+1} = -u$, so that z becomes type $A_{i+1}^{a_i(z)+1} = A_{i+1}^{a_{i+1}(z)}$. (II) If $b_i(z) \neq b_{i-1}(z)$ we proceed similarly, first noting that $a_{i+1}(z)=a_i(z)$. By Definition 4.6, $E_i = E_{i,u}$ where $u=(-1)^{a_i(z)+b_{i-1}(z)}$ and in

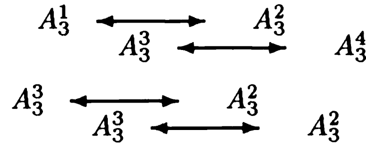


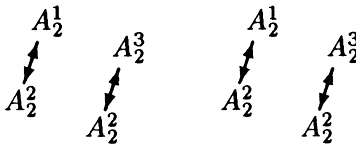
(a) After initial step and C_1.

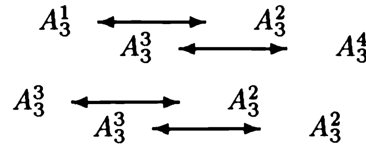


(b) After P_0. Arrows describe E_0.









(c) After P_1. Arrows describe E_1.

(d) After P_2. Arrows describe E_2.

(e) After P_3. Arrows describe E_3.

(f) The last step of Q.

FIG. 10. *Subcube states in the token exchange for the* 9-*cube.*

applying Lemma 4.2 we find that $(-1)^{b_i(z)+a_i(z)+1} = (-1)^{b_{i-1}(z)+a_i(z)} = u$, so that z becomes $A_{i+1}^{a_i(z)} = A_{i+1}^{a_{i+1}(z)}$. $\square$

Phase Q begins with every subcube z of type $A_D^{a_D(z)}$ and it proceeds to make them all $A_D^{D+1} = A_D^d$. (Recall that $d = D + 1$.) Q uses both C_k and copying steps and is ad hoc in nature, having been defined only in specific cases given below.

Final step. After Q, each subcube is type A_D^d and can be finished in one step by having all $L_{x,n}$ with $P(n) = (-1)^d$ transmit along some dimension. It can now be seen that the total number of steps is $3D + 2+$ the number of steps in Phase Q. We now proceed to examine Q.

Prior to Q we have one subcube A_D^{D+1}, namely, $x = 2^{D+1} - 1$. This is the only x for which $a_D(x) = D + 1$. We now use binary notation: $x = 111 \cdots 111 = 1^D$, where we will always have D bits. There are D subcubes A_D^D because there are D values x for which $a_D(x) = D$; they are $01^{D-1}, 0^2 1^{D-2}, \cdots, 0^D$. In general, there are $\binom{D}{k}$ values x for which $a_D(x) = D + 1 - k$. To see this, observe that $a_D(x)$ is $D+1$ minus the number of values i for which $b_i(x)$ differs from $b_{i-1}(x)$; for each x consider the set $S_x = \{i \mid b_i(x) \neq b_{i-1}(x)\}$; then whenever $|S_x| = k$, we will have $a_D(x) = D + 1 - k$; and finally it is easy to see that there are $\binom{D}{k}$ such sets S_x.

Phase Q generally proceeds this way. Copy subcubes of type A_D^{D+1} to neighboring subcubes at each step. Apply C_k to subcubes of type A_D^k with $k \leq D$, except when the subcube is the target of a copying of a subcube of type A_D^{D+1}. Q terminates when all subcubes are of type A_D^{D+1}. We must choose which neighbors to use during each copying operation so as to minimize the total number of steps of Q, by avoiding neighbors that are able to become type A_D^{D+1} soon enough on their own using the C_k transformations, and we do this only for three specific cases.

First, for $D < 4$, one cannot do better than simply copy the one original A_D^{D+1} subcube to all subcube sites in D steps, for a total of $4D + 2 = 2(D + d)$ steps.

Second, for $D = 4$ we have a type A_4^5 subcube at 1111; copy it to 1100, 1101, and 1110 in two steps. And we have type A_4^4 subcubes at 0111, 0011, 0001, and 0000; apply C_4 to them in two steps. Then we have altogether eight A_4^5 subcubes arranged so that it is possible to copy them to the remaining eight sites in one step. Thus Q takes three steps and the complete algorithm takes $17 = \frac{17}{9}(D + d)$ steps.

Figure 10 shows the states of the subcubes at several intermediate stages of the operation of this algorithm for $D = 4$.

Third, for $D = 8$, we start with one A_8^9 cube, 8 A_8^8 cubes, and 28 A_8^7 cubes. Particular choices can be made which we will only summarize here so that in 6 steps we copy 64 images of the A_8^9, we apply C_8 to the 8 A_8^8 subcubes followed by 4 copying steps to produce 128 more A_8^9 subcubes, and we apply C_7 and then C_8 to 16 of the A_8^7 subcubes followed by 2 copying steps to produce 64 more A_8^9 subcubes. (The A_8^9 at 11111111 is copied to all locations of the form $1{*}{*}{*}1{*}{*}{*}$ in 6 steps, where $*$ can be either 0 or 1; the A_8^8 subcubes are at $01111111, \cdots, 00000000$, and after converting them to A_8^9 we copy them to all locations of the form $0{*}{*}{*}{*}{*}{*}{*}$ in four steps; we use A_8^7 subcubes at locations $10000xyz$, $11000xyz$, $11100xyz$, and $11110xyz$ where $xyz = \{000, 001, 011, 111\}$, and after converting them to A_8^9 we copy them, respectively, to all sites of the form $100{*}0{*}{*}{*}$, $110{*}0{*}{*}{*}$, $1{*}100{*}{*}{*}$, and $1{*}110{*}{*}{*}$ in 2 steps, which puts them at all locations of the form $1{*}{*}{*}0{*}{*}{*}$.) This gives us 256 A_8^9 subcubes in all, one at each site. Thus for $D = 8$, Q can be done in 6 steps and the complete algorithm takes $32 = \frac{32}{17}(D + d)$ steps.

We summarize these results in the following theorem.

THEOREM 4.2. *The token exchange problem under the half-duplex pairwise model can be solved on an n-dimensional hypercube in fewer than* $2n$ *steps if* $n \geq 9$.

Proof. To solve the problem in $2n - 1$ steps, choose a nine-dimensional subcube, in $n - 9$ steps send all tokens to that subcube, perform the exchange on the subcube in 17 steps, and then in $n - 9$ final steps send all tokens to all sites. □

4.4. Asymptotic behavior. It is likely that as $D \to \infty$ it is possible to obtain better and better solutions, although we have no general method for making good choices in Q. However, the result below shows that $D = 4$ is already close to the best possible.

THEOREM 4.3. *The complete solution of the $(D+d)$-dimensional hypercube token exchange problem using P and Q takes at least $\frac{1}{2}(6-\lg 5)(D+d) \approx 1.85(D+d)$ steps.*

Proof. We estimate the number p of A_D^{D+1} subcubes that Q is able to generate in $2k$ steps. In $2k$ steps it can generate 2^{2k} from the one A_D^{D+1}, 2^{2k-2} from each of the $\binom{D}{1}$ A_D^D subcubes, 2^{2k-4} from each of the $\binom{D}{2}$ A_D^{D-1} subcubes, and so on. Thus we have

$$\begin{aligned} p &= 2^{2k} + 2^{2k-2}\left(\begin{array}{c} D \\ 1 \end{array}\right) + \cdots + \left(\begin{array}{c} D \\ k \end{array}\right) \\ &\leq 4^k\left(1+\frac{1}{4}\right)^D \\ &= 4^{k-D}5^D. \end{aligned}$$

Since Q must yield 2^D subcubes of type A_D^{D+1}, we must have $4^{k-D}5^D \geq 2^D$ so that

$$2k \geq (3 - \lg 5)D.$$

The number of steps in the complete solution is then at least

$$\begin{aligned} 3D + 2 + (3 - \lg 5)D &= \left(\frac{6 - \lg 5}{2}\right) 2D + 2 \\ &> \left(\frac{6 - \lg 5}{2}\right) 2D + \left(\frac{6 - \lg 5}{2}\right) \\ &= \left(\frac{6 - \lg 5}{2}\right)(D + d) \\ &\approx 1.850(D + d). \end{aligned}$$

□

If we seek the minimal value of K for which we can find a solution in $K(D + d)$ steps, we see that for $D < 4$ we have $K = 2$, for $D = 4$ we have $K = \frac{17}{9} \approx 1.889$, for $D = 8$ we have $K = \frac{32}{17} \approx 1.882$, and as $D \to \infty$ we can expect K to become smaller. However, K is bounded below by ≈ 1.850, so $D = 4$ already gives close to the best rate of solution possible using this approach. We summarize our best known K as follows.

THEOREM 4.4. *The token exchange problem under the half-duplex pairwise model can be solved on a $(17s+t)$-dimensional hypercube in $32s+2t$ steps, or in other words in $\approx 1.88n$ steps on an n-dimensional cube.*

Proof. First send all tokens to a $17s$-dimensional subcube in t steps, solve the problem in $32s$ steps on the subcube, and finally send the tokens back to all sites in t steps. For the $17s$-dimensional subcube, treat it as the Cartesian product of a $17(s-1)$-cube and a 17-cube, solve the problem on all 17-cubes in 32 steps, and recursively solve the problem on the $17(s-1)$-cubes in $32(s-1)$ steps. □

5. Discussion. For the d-dimensional hypercube under the half-duplex pairwise model, there remains a gap between the lower bound of $1.44d$ that applies to any topology [3] and the upper bound of $1.88d$. Narrowing this gap seems to be a hard problem. The algorithm of §4 relies on moves that either double the number of tokens

present at a site or that copy token sets from one site to a new site that may as well not have contained any tokens at all before the copying. Using the same methods that were used to establish the lower bound of $1.44d$, one can show that if only these kinds of moves are used, then the lower bound is $(\lg 3)d \approx 1.58d$, and that if half the moves are doubling ones and half are copying ones, then the lower bound is $(4/\lg 5)d \approx 1.72d$. Thus the fact that the C_k sequences use copying and doubling moves increases the lower bound from $1.44d$ to $1.58d$, the fact that they use equal measures of each raises it further to $1.72d$, and the presence of the E_i moves raises it to $1.85d$. The use of copying and doubling and of the E_i all solve problems created by the geometry of the hypercube, so it will not be easy to make improvements on this algorithm. Attempts to emulate the Fibonacci algorithm that solves the token exchange problem in $1.44 \lg N$ steps on a complete graph of N nodes have been thwarted by the interconnect pattern of the hypercube.

Acknowledgment. The referee's thoroughness and helpfulness are gratefully acknowledged.

REFERENCES

[1] A. Bagchi, S. L. Hakimi, J. Mitchem, and E. Schmeichel, *Parallel algorithms for gossiping by mail*, Inform. Process. Lett., 34 (1990), pp. 197–202.

[2] S. M. Hedetniemi, S. T. Hedetniemi, and A. L. Liestman, *A survey of gossiping and broadcasting in communications networks*, Networks, 18 (1988), pp. 320–349.

[3] D. W. Krumme, K. N. Venkataraman, and G. Cybenko, *Gossiping in minimal time*, SIAM J. Comput., 21 (1992), pp. 111–139.

[4] J. E. McCanna, *Orientations of the n-cube with minimum diameter*, Discrete Math., 68 (1988), pp. 309–310.

SIAM J. COMPUT.
Vol. 21, No. 2, pp. 381–406, April 1992

© 1992 Society for Industrial and Applied Mathematics
012

RECOGNIZING P_4-SPARSE GRAPHS IN LINEAR TIME*

BEVERLY JAMISON† AND STEPHAN OLARIU‡

Abstract. A graph G is P_4-sparse if no set of five vertices in G induces more than one chordless path of length three. P_4-sparse graphs generalize both the class of cographs and the class of P_4-reducible graphs. One remarkable feature of P_4-sparse graphs is that they admit a tree representation unique up to isomorphism. It has been shown that this tree representation can be obtained in polynomial time. This paper gives a linear time algorithm to recognize P_4-sparse graphs and shows how the data structures returned by the recognition algorithm can be used to construct the corresponding tree representation in linear time.

Key words. cographs, P_4-sparse graphs, linear-time algorithms, optimal algorithms

AMS(MOS) subject classifications. 05, 68

1. Introduction. Due to their wide applications in communications, transportation, VLSI design, program optimization, database design, and other areas of computer science and engineering, graph problems often require fast solutions. It is well known, however, that many interesting problems in graph theory are NP-complete on general graphs. Fortunately, in practical applications one rarely has to contend with general graphs: typically, a careful analysis of the problem at hand reveals sufficient structure to limit the graphs under investigation to a restricted class.

Quite often, real-life applications suggest the study of graphs that feature some "local density" properties. In particular, graphs that are unlikely to have more than a few chordless paths of length three (also referred to as P_4's) appear in a number of contexts. Examples include examination scheduling and semantic clustering of index terms (see [4]).

In examination scheduling, for example, a *conflict graph* is readily constructed: the vertices represent different courses offered, while courses x and y are linked by an edge if and only if some student takes both of them. (In the weighted version, the weight of edge xy stands for the number of students taking both x and y.) Clearly, in any coloring of the conflict graph, vertices that are assigned the same color correspond to courses whose examinations can be held concurrently. It is usually anticipated that very few paths of length three will occur in the conflict graph.

In the second application, we construct a graph whose vertices are the index terms; an edge occurs between two index terms to denote self-referencing or semantic proximity. Again, very few P_4's are expected to occur. These applications have motivated both the theoretical and algorithmic study of the classes of cographs [2]–[4], [9], [10], which contain no induced P_4's, and P_4-reducible graphs [6], defined as those graphs in which no vertex belongs to more than one P_4.

The study of cographs and P_4-reducible graphs led naturally to constructive characterizations that implied tree representations unique up to isomorphism (the interested reader is referred to [4], [6], [7] for details). The constructions were such that the tree representation could be obtained in polynomial time. Further study yielded a linear-time recognition algorithm that also enabled the construction of the tree in linear time.

*Received by the editors September 8, 1989; accepted for publication (in revised form) May 14, 1991.

†Department of Mathematics and Computer Science, University of Hartford, Hartford, Connecticut 06117.

‡Department of Computer Science, Old Dominion University, Norfolk, Virginia 23529-0162. The work of this author was supported in part by National Science Foundation grant CCR-8909996.

The class of P_4-sparse graphs was first introduced by Hoáng [5] in his doctoral dissertation: these are the graphs for which every set of five vertices induces at most one P_4. It is easy to see that the P_4-sparse graphs generalize both the cographs and the P_4-reducible graphs. In [8], Jamison and Olariu gave several structural theorems for P_4-sparse graphs, including a constructive characterization asserting that the P_4-sparse graphs are exactly the graphs constructible from single-vertex graphs by three graph operations. This result implies that P_4-sparse graphs have a unique tree representation up to isomorphism.

The purpose of this paper is to provide a linear-time recognition algorithm for P_4-sparse graphs. This algorithm is incremental in nature: the vertices are processed one at a time, with a determination made at each iteration as to whether the subgraph induced by the vertices processed so far is P_4-sparse.

The algorithm returns a maximum size induced P_4-free subgraph and a structure containing the remaining vertices. These two structures can be used to construct the tree representation in linear time.

The paper is organized as follows: §2 provides the background information on the structure of P_4-sparse graphs and the main theorem on which the linear-time recognition is based; §3 presents the data structures and procedures used in the accept or reject decision; §4 gives the details of the update of the data structures that occurs whenever the new graph is P_4-sparse; §5 presents the construction of the tree representation of P_4-sparse graphs; finally, §6 summarizes the results and poses a number of open problems.

2. Background and terminology. All the graphs in this work are finite, with neither loops nor multiple edges. In addition to standard graph-theoretical terminology compatible with Bondy and Murty [1], we use some new terms that we are about to define.

Let $G = (V, E)$ be an arbitrary graph. For a vertex x of G, we let $N_G(x)$ denote the set of all the vertices of G that are adjacent to x: we assume adjacency to be nonreflexive, and so x does not belong to $N_G(x)$; we let $d_G(x)$ stand for $|N_G(x)|$. For a subset S of the vertex-set of G, we let G_S stand for the subgraph of G induced by S. If a vertex x is nonadjacent to a vertex y, we shall say that x *misses* y. (Similarly, y *misses* x.) A vertex z is said to *distinguish* between vertices u and v, whenever z misses precisely one of u, v.

To simplify the notation, a P_4 with vertices a, b, c, d and edges ab, bc, cd will be denoted by $abcd$. In this context, the vertices a and d are referred to as *endpoints* while b and c are termed *midpoints* of the P_4. Consider a P_4 in G induced by $A = \{a, b, c, d\}$. A vertex x outside A is said to have a *partner* in A if x together with three vertices in A induces a P_4 in G. Given an induced subgraph H of G and a vertex x outside H, we say that x is *neutral* with respect to H if x has a partner in no P_4 in H. In the remaining part of this work we shall often associate, in some way, rooted trees with graphs. In this context, we shall refer to the vertices of trees as *nodes*. For a node w in a tree T, we let $p(w)$ stand for the parent of w in T.

To make this paper self-contained, we shall review some of the properties of cographs and P_4-sparse graphs.

To begin, Lerchs [9] showed how to associate with every cograph G a unique tree $T(G)$ called the *cotree* of G, defined as follows:

- Every internal node, except possibly for the root, has at least two children.
- The internal nodes are labeled by either 0 (0-nodes) or 1 (1-nodes) in such a way that the root is always a 1-node, and such that 1-nodes and 0-nodes alternate along every path in $T(G)$ starting at the root.
- The leaves of $T(G)$ are precisely the vertices of G, such that vertices x and y are adjacent in G if and only if the lowest common ancestor of x and y in $T(G)$ is a

1-node.

In our characterization of P_4-sparse graphs, we make use of the properties of a special graph that we are about to define. A graph G is termed a *spider* if the vertex-set V of G admits a partition into disjoint sets S, K, R such that:
(s1) $|S| = |K| \geq 2$, S is stable, K is a clique;
(s2) Every vertex in R is adjacent to all the vertices in K and misses all the vertices in S;
(s3) There exists a bijection $f : S \to K$ such that either

$$N_G(s) \cap K = \{f(s)\} \text{ for all vertices } s \text{ in } S,$$

or

$$N_G(s) \cap K = K - \{f(s)\} \text{ for all vertices } s \text{ in } S.$$

It is easy to see that the complement of a spider is also a spider. For further reference we make the following observation about the P_4's in a spider.

OBSERVATION 1. Let G be a spider. Every P_4 in G has vertices in $K \cup S$ or in R only. Furthermore, if a P_4 has vertices in $K \cup S$, then it is induced by a set of the form $\{x, y, f(x), f(y)\}$ with distinct x, y in S.
(Follows directly from (s1)–(s3).)

In [8], Jamison and Olariu prove the following characterization of P_4-sparse graphs.

PROPOSITION 2.1 (Theorem 1 in [8]). *For a graph G, the following conditions are equivalent:*

(i) *G is a P_4-sparse graph;*
(ii) *for every induced subgraph H of G, exactly one of the following statements is satisfied:*
(1) *H is disconnected;*
(2) *$\overline{H}$ is disconnected;*
(3) *H is isomorphic to a spider.*

Lerchs [10] proved that the cographs are precisely the graphs obtained from single-vertex graphs by a finite sequence of ⓪ and ① operations defined as follows. Given disjoint graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, define:

- G_1 ⓪ $G_2 = (V_1 \cup V_2, E_1 \cup E_2)$;
- G_1 ① $G_2 = (V_1 \cup V_2, E_1 \cup E_2 \cup \{xy | x \in V_1, y \in V_2\})$.

It is easy to see that the operations ⓪ and ① reflect (1) and (2), respectively, in condition (ii) of Proposition 2.1. For the purpose of constructing the P_4-sparse graphs, we need to introduce a third graph operation to reflect (3).

Consider disjoint graphs $G_1 = (V_1, \emptyset)$ and $G_2 = (V_2, E_2)$ with $V_2 = \{v\} \cup K \cup R$ such that

(j1) $|K| = |V_1| + 1 \geq 2$;
(j2) K is a clique;
(j3) Every vertex in R is adjacent to all the vertices in K and nonadjacent to v;
(j4) There exists a vertex v' in K such that $N_{G_2}(v) = \{v'\}$ or $N_{G_2}(v) = K - \{v'\}$.

Choose a bijection $f : V_1 \to K - \{v'\}$ and define

$$G_1 \text{②}\, G_2 = (V_1 \cup V_2, E_2 \cup E') \tag{1}$$

with

$$E' = \begin{cases} \{xf(x) | x \in V_1\} & \text{whenever } N_{G_2}(v) = \{v'\}, \\ \{xz | x \in V_1, z \in K - \{f(x)\}\} & \text{whenever } N_{G_2}(v) = K - \{v'\}. \end{cases}$$

PROPOSITION 2.2 (Theorem 2 in [8]). *G is a P_4-sparse graph if and only if G is obtained from single-vertex graphs by a finite sequence of operations ⓪ ,① ,② .*

The following natural observation follows directly from Proposition 2.2.

OBSERVATION 2. Let G be a P_4-sparse graph. If G (respectively, $\overline{G}$) is disconnected with components $G_1, G_2, \cdots, G_p$ ($p \geq 2$), then we can write $G = G_1 \textcircled{0}\ (\textcircled{1})\ G_2 \textcircled{0}\ (\textcircled{1}) \cdots \textcircled{0}\ (\textcircled{1})\ G_p$.
(Observation 2 follows from Proposition 2.2 by a trivial inductive argument.)

Propositions 2.1 and 2.2 and Observation 2 combined suggest a natural way of associating with every P_4-sparse graph G a tree $T(G)$ (called the *ps-tree* of G), as described by the following recursive procedure.

Procedure Build_tree(G);
{Input: a P_4-sparse graph $G = (V, E)$;
Output: the ps-tree $T(G)$ corresponding to G.}
begin
    if $|V| = 1$ **then**
        return the tree $T(G)$ consisting of the unique vertex of G;
    if G (respectively, $\overline{G}$) is disconnected **then begin**
        let $G_1, G_2, \cdots, G_p$ (respectively, $\overline{G_1}, \overline{G_2}, \cdots, \overline{G_p}$) ($p \geq 2$) be the components of G (respectively, $\overline{G}$);
        let $T_1, T_2, \cdots, T_p$ be the corresponding ps-trees rooted at $r_1, r_2, \cdots, r_p$;
        return the tree $T(G)$ obtained by adding $r_1, r_2, \cdots, r_p$ as children of a node labeled 0 (1);
        end
    else begin {now both G and $\overline{G}$ are connected}
        write $G = G_1 \textcircled{2}\ G_2$ as in (1);
        let T_1, T_2 be the corresponding ps-trees rooted at r_1 and r_2;
        return the tree $T(G)$ obtained by adding r_1, r_2 as children of a node labeled 2
        end
end; {Build_tree}

As it turns out (see [8]) the ps-tree of a P_4-sparse graph G is unique up to isomorphism. In the remainder of this section, we introduce two theorems which lay the foundation for our recognition algorithm: Theorem 2.3 characterizes the P_4-sparse graphs in terms of their P_4's; Theorem 2.5 gives the requirements for maintaining the P_4-sparse property when introducing a new vertex into a P_4-sparse graph.

Let G be an arbitrary graph. We shall say that G has a *special partition* if there exists a family $\Sigma = \{S_1, S_2, \cdots, S_q\}$ ($q \geq 1$) of disjoint stable sets of G with $|S_i| \geq 2$ ($1 \leq i \leq q$) and an injection

$$f : \bigcup_{i=1}^{q} S_i \to V - \bigcup_{i=1}^{q} S_i$$

such that the following two conditions are satisfied:
(sp1) K_i=f(S_i) is a clique for all $i = 1, 2, \cdots, q$;
(sp2) A set A of vertices induces a P_4 in G if and only if there exists a subscript i ($1 \leq i \leq q$) and distinct vertices x, y in S_i such that $A = \{x, y, f(x), f(y)\}$.

We are now in a position to state our first result which provides a new characterization of P_4-sparse graphs.

THEOREM 2.3. *For a graph G the following statements are equivalent:*
(i) *G is P_4-sparse;*

(ii) *G is a cograph or G has a special partition.*

Proof. The implication (ii)→(i) is easy: if G is a cograph, then there is nothing to prove; otherwise, we only need observe that no set of five vertices in G can contain the union of two distinct sets of the form $\{x, y, f(x), f(y)\}$. To prove the implication (i)→(ii), we proceed by induction. Assuming the implication true for all P_4-sparse graphs with fewer vertices than G, we only need prove that G itself satisfies (ii) whenever G is P_4-sparse. We first observe that if V can be partitioned into disjoint sets V_1, V_2 such that G_{V_1} and G_{V_2} satisfy (ii) and no P_4 in G has vertices from both G_{V_1} and G_{V_2}, then G is either a cograph or it has a special partition.

If G or $\overline{G}$ is disconnected, then each component satisfies (ii) by the induction hypothesis. Otherwise, by Proposition 2.1, G is a spider. Now $S \cup K$ and R each induce P_4-sparse graphs with fewer vertices than G. Further, no P_4 in G has vertices from both graphs. Thus, we are done by the induction hypothesis. □

It is easy to see that conditions (sp1) and (sp2) above guarantee that if a graph G has a special partition, then the sets S_i and K_i are unique. The following observation shows how the sets S_i, K_i $(1 \leq i \leq q)$ in a special partition of a P_4-sparse graph G can be used to verify the neutrality of a vertex with respect to G.

OBSERVATION 3. A vertex $x \notin G$ is neutral with respect to G if and only if for every subscript i $(1 \leq i \leq q)$ the following two conditions hold:

(i) if x is adjacent to a vertex in S_i, then x is adjacent to all the vertices in $S_i \cup K_i$;

(ii) if a vertex x is adjacent to a vertex in K_i, then x is adjacent to all the vertices in K_i.

(Observation 3 follows trivially from the definition of neutrality together with Theorem 2.3.)

We will now define a special subgraph of a P_4-sparse graph that is central to our incremental recognition algorithm for P_4-sparse graphs. Let $G = (V, E)$ be a P_4-sparse graph. The *canonical cograph* $C(G)$ associated with G is the induced subgraph of G obtained by the following procedure.

```
Procedure Greedy(G);
{Input: a P4-sparse graph G;
Output: the canonical cograph C(G)}
begin
    C(G) ← G;
    while there exist P4's in C(G) do begin
        pick a P4 uvxy in C(G);
        pick z arbitrarily in {u, y};
        C(G) ← C(G) − {z}
    end;
    return(C(G))
end; {Greedy}
```

An easy inductive argument shows that for every induced subgraph H of G which satisfies (3) in (ii) of Proposition 2.1, the procedure Greedy removes all the vertices in S_i $(1 \leq i \leq q)$, except for an arbitrary one. The fact that the graph $C(G)$ returned by Greedy is a cograph, as well as the fact that $C(G)$ is as large as possible is immediate. The uniqueness implied by the term *canonical* is justified by the following stronger result.

PROPOSITION 2.4 (Theorem 3 in [8]). *For a graph G with no induced C_5, the following statements are equivalent:*

(i) *G is P_4-sparse;*

(ii) *For every induced subgraph H of G, $C(H)$ is unique up to isomorphism.*

We are now in a position to give the result which is the basis for the recognition algorithm.

THEOREM 2.5. *If G is a P_4-sparse graph, then $G + x$ is P_4-sparse if and only if the following conditions are satisfied:*

(2) *x is neutral with respect to G;*

(3) *x belongs to at most one P_4 in $C(G) + x$.*

Proof. To prove the "only if" part, assume that $G + x$ is a P_4-sparse graph. Since (2) is trivially satisfied, we only need prove that (3) holds. In fact, we shall prove the following stronger result.

(4) *For every induced cograph H of G, x belongs to at most one P_4 in $H + x$.*

To see that (4) is true, we use induction on the size of G. If $G + x$ or $\overline{G+x}$ is disconnected, then we are done by the induction hypothesis applied to the component containing x. We may, therefore, assume that both $G + x$ and $\overline{G+x}$ are connected. If G is not P_4-free, then $H + x$ is a P_4-sparse graph with strictly fewer vertices than $G + x$. Now we are done by the induction hypothesis applied to $H + x$.

Finally, we may assume that G is P_4-free. If $G + x$ is P_4-free, then there is nothing to prove. Otherwise, by virtue of Proposition 2.1, $G+x$ is a spider. The vertices of $G+x$ can be split into S, K, and R as in (s1)–(s3). Since G contains no P_4, x belongs to $K \cup S$. If x belongs to two distinct P_4's in $G+x$, then by Theorem 2.3, these P_4's are induced by the sets $\{x, x', y, f(y)\}$ and $\{x, x', z, f(z)\}$ with $y \neq z$, and with $x = f(x')$ or $x' = f(x)$, depending on whether or not $x \in K$.

But now, Theorem 2.3 guarantees that $\{y, z, f(y), f(z)\}$ induces a P_4 in G, contradicting the assumption that G is P_4-free. Thus (4) must be true.

To prove the "if" part, let G be a P_4-sparse graph and suppose that both (2) and (3) are satisfied, yet $G + x$ is not P_4-sparse. There must exist a set of five vertices in $G + x$ inducing two distinct P_4's. We let B stand for such a set containing as few vertices from $G - C(G)$ as possible. Clearly, since G is P_4-sparse, we have $x \in B$. Since (3) is satisfied, B must contain vertices from $G - C(G)$. Let a be such a vertex.

Now the definition of the canonical cograph $C(G)$ implies the existence of vertices b, c, d in $C(G)$ such that $A = \{a, b, c, d\}$ induces a P_4 in G with edges ab, bc, cd.

Our arguments rely on the following simple observation whose justification amounts to a straightforward case-by-case analysis.

OBSERVATION 4. Let $G = (V, E)$ be an arbitrary graph. If a set A induces a P_4 in G, then the subgraph of G induced by $A \cup \{x, y\}$ is P_4-sparse whenever $x, y \in V - A$ have no partners in A.

By Observation 4, we must have $|A \cap B| \leq 2$. By (2), no vertex in $B - A$ has a partner in A. This implies that

$$(5)\quad N_G(a) \cap (B - A) = N_G(d) \cap (B - A) \text{ and } N_G(b) \cap (B - A) = N_G(c) \cap (B - A).$$

By (5), $|A \cap B| = 1$ leads to an immediate contradiction: the set $B - \{a\} \cup \{d\}$ induces two distinct P_4's in G, contradicting our choice of B.

Now we may assume that $|A \cap B| = 2$. But now if $A \cap B = \{a, b\}$, then by (5) the set $(B - \{a, b\}) \cup \{c, d\}$ contradicts our choice of B; if $A \cap B = \{a, c\}$, then the set $(B - \{a, c\}) \cup \{b, d\}$ contradicts our choice of B. Finally, if $A \cap B = \{a, d\}$, then by (5), $B - \{a\}$ must induce a P_4, and either b or c must have a partner in this P_4. This completes the proof of Theorem 2.5. □

3. Algorithms I: Recognition. We shall use the criteria obtained in Theorem 2.5 to develop a linear-time incremental algorithm to recognize P_4-sparse graphs. In the process, we maintain two data structures describing the portion of the graph successfully processed. One structure represents the canonical cograph; the other contains all vertices which belong to some P_4 in the graph. In case the graph is P_4-sparse, it turns out that these two structures can be used to construct the ps-tree of the graph in linear time.

To define our data structures and outline our recognition algorithm for P_4-sparse graphs, consider an arbitrary graph G. We assume that we have already processed a nonempty induced P_4-sparse subgraph H of G. (Note that such a subgraph H can always be found; in fact, the subgraph induced by a subset of at most four vertices in G is a P_4-sparse graph.) $T(H)$ contains the canonical cotree of H, i.e. the cotree representation of the canonical cograph $C(H)$.

The insight provided by Theorem 2.3 motivates our method of storing the P_4's in the special partition of H. More precisely, we use the array $SK(H)$ to store the sets S_i and K_i of the special partition of H. (Naturally, when we write $SK(i)$ we mean the ith element of the array SK.) As vertices are added to $SK(H)$, $SKsize(i)$ keeps track of the cardinality of S_i and $SKadj(i)$ keeps track of the cardinality of $N_{K_i}(s)$ for $s \in S_i$. Vertices in $SK(H)$ will be stored as ordered pairs $(s, g(s))$ where $g(s)$ is the unique vertex in K_i adjacent to s if $SKadj(i) = 1$ and the unique vertex in K_i nonadjacent to s if $SKadj(i) > 1$.

We also maintain, for every vertex v in G, pointers to the location, if any, of that vertex in $SK(H)$ and $T(H)$. Specifically, $Tree(v)$ points to the location in $T(H)$; $SKindex(v)$ gives the index of the ordered pair containing v (if any) in $SK(H)$; $SKtype(v)$ is either "S" or "K." Throughout the algorithmic sections, we shall assume that all data are initialized to zero or null, as appropriate.

The actual recognition algorithm consists of the following two stages.

Algorithm Recognize(G);
Stage 1. {Initialization}
 $H \leftarrow \{v_1, v_2\}$;
 construct the cotree $T(H)$ rooted at R;
 $SK(H) \leftarrow \emptyset$;
Stage 2. {Incrementally process the remaining vertices in $G - H$, as follows}
 Step 2.0 pick x in $G - H$;
 Step 2.1 if x is not neutral with respect to H then return("no");
 Step 2.2 if x belongs to more than one P_4 in $C(H) + x$ then return("no");
 Step 2.3 $H \leftarrow H + x$; update $T(H)$ and $SK(H)$.

In the incremental stage of the algorithm, we first determine whether or not $H + x$ is P_4-sparse. If it is, then we update the data structures to represent $T(H + x)$ and $SK(H + x)$. When the incremental stage is complete, we have either rejected the graph or constructed $T(G)$ and $SK(G)$. Theorem 2.5 guarantees that to determine whether or not $H + x$ is P_4-sparse, we need only verify that (2) and (3) are satisfied.

Observation 3 gives the criteria for a vertex to be neutral with respect to a P_4-sparse graph. In order to verify (2) in time proportional to $d_H(x)$, we shall examine each neighbor of x and maintain counters of how many vertices in each set S_i and each set $S_i \cup K_i$ are processed. We shall also maintain test bits $Ktest(i)$ and $SKtest(i)$. The procedure UpdateP_4 increments the counters and sets the appropriate test bit whenever a counter reaches the cardinality of the set with which it is associated. UpdateP_4 also builds a list, $SKlist$, of neighbors of x which belong to some P_4 in H. The procedure TestNeutral

processes each vertex in $SKlist$ by checking the appropriate test bit and decrementing the associated counter. If any test bit indicates the criteria are not met, the graph is rejected. TestNeutral also resets a test bit whenever the associated counter reaches 0. All data are reinitialized in the process. Both UpdateP_4 and TestNeutral process each neighbor of x at most once and each iteration of the main loop is completed in constant time. Thus, we can state

(6) *Step* 2.1 *of Stage* 2 *is performed in time bounded by* $d_G(x)$.

The procedure UpdateP_4 is implemented as follows.

Procedure Update$P_4(x)$;
1. **begin**
2. **for** each $v \in N_H(x)$ such that $SKindex(v) > 0$ **do begin**
3. $i \leftarrow SKindex(v)$;
4. $SKcount(i) \leftarrow SKcount(i) + 1$;
5. **if** $SKcount(i) = SKsize(i)$ **then** $SKtest(i) \leftarrow 1$;
6. **if** $SKtype(v) =$ "K" **then begin**
7. $Kcount(i) \leftarrow Kcount(i) + 1$;
8. **if** $Kcount(i) = Ksize(i)$ **then** $Ktest(i) \leftarrow 1$;
9. **end**
10. **end**
11. **end**; {UpdateP_4}

The details of procedure TestNeutral are spelled out as follows.

Procedure TestNeutral(x);
1. **begin**
2. **for** each vertex v in $SKlist$ **do begin**
3. $i \leftarrow SKindex(v)$;
4. $t \leftarrow SKtype(v)$;
5. **if** $SKtest(i) \neq 1$ and t='S' **then** return "no";
6. $SKcount(i) \leftarrow SKcount(i) - 1$;
7. **if**$SKcount(i) = 0$ **then** $SKtest(i) \leftarrow 0$;
8. **if** t="K" **then begin**
9. **if** $Ktest(i) \neq 1$ **then** return "no";
10. $Kcount(i) \leftarrow Kcount(i) - 1$;
11. **if** $Kcount(i) = 0$ **then** $Ktest(i) \leftarrow 0$
12. **end**
13. **end**
14. **end**; {TestNeutral}

An algorithm for testing (3) in time proportional to $d_H(x)$ relies on a generalization of the marking scheme developed by Corneil, Perl, and Stewart [4]. For convenience, we borrow their notation relevant to the marking scheme.

For a node u in the canonical cotree $T(H)$, rooted at R, we let $d(u)$ stand for the number of children of u; $md(u)$ represents the current number of marked, and subsequently unmarked children of u. (Initially, $md(u)$ is 0 for all the nodes u in $T(H)$; when u is unmarked, $md(u)$ is reset to 0.) A marked 1-node of $T(H)$ is said to be *properly marked* whenever $md(u) = d(u) - 1$; otherwise it will be termed *improperly marked*.

The procedure Mark using the adjacency information of a new vertex x performs the following actions:

- marks, and subsequently unmarks, as appropriate, certain nodes of $T(H)$;
- adds marked but not subsequently unmarked nodes of $T(H)$ to one or the other of the linked lists M_0 (containing marked 0-nodes), M_1 (containing improperly marked 1-nodes) or M_2 (containing properly marked 1-nodes).

It turns out that (3) is satisfied if and only if all vertices remaining marked at the termination of Mark lie on a single path (to the root) with special properties which will be detailed shortly. In order to test (3), we first mark the cotree with respect to x. Then we identify a vertex α which, if (3) holds, will be the lowest vertex still marked. Finally, we test the path from α to the root for the desired properties.

Thus Step 2.2 can be further refined as follows:

Step 2.2. {if x belongs to more than one P_4 in $C(H) + x$ then return("no")}
Step 2.2.1. Mark(x);
Step 2.2.2. Find α;
Step 2.2.3. If the path in $T(H)$ joining R and α is not admissible then return("no");

We will give the procedures used to accomplish each of the three substeps and the appropriate proofs.

Procedure Mark(x);

```
0.  begin
1.      M0 ← M1 ← M2 ← ∅; c0 ← c1 ← c2 ← 0;
2.      for each v in N_H(x) do
3.          if v is a leaf in T(H) then mark v;
4.          for each marked node u in T(H) do
5.              if d(u) = md(u) then begin
6.                  unmark u;
7.                  md(u) ← 0;
8.                  if u ≠ R then begin
9.                      w ← p(u);
10.                     mark w;
11.                     md(w) ← md(w) + 1;
12.                     add u to the list of marked and subsequently unmarked
                         children of w
13.                     end
14.                 end
15.             else {now d(u) ≠ md(u), and so u is marked but not unmarked}
16.                 case label(u) of
17.                     0:    begin {u is a marked 0-node}
18.                               c0 ← c0 + 1;
19.                               M0 ← M0 ∪ {u}
20.                           end;
21.                     1:    begin
22.                               if md(u) ≠ d(u)−1 then begin {u is improperly marked}
23.                                   c1 ← c1 + 1;
24.                                   M1 ← M1 ∪ {u}
25.                                   end;
26.                               else begin {now u is a properly marked 1-node}
27.                                   c2 ← c2 + 1;
```

```
28.                     M_2 ← M_2 ∪ {u}
29.                     end;
30.                 end
31.         endcase;
32.     if (c_0 + c_1 + c_2 > 0) and d(R)=1 then markR
33. end; {Mark}
```

OBSERVATION 5. Mark(x) runs in $d_G(x)$ time.

To justify this claim, note that since all the internal nodes of $T(H)$, with the possible exception of the root, have at least two children, the number of nodes that are marked (and possibly subsequently unmarked) is bounded by $d_G(x)$. Hence, $|M_0 \cup M_1 \cup M_2|$ is bounded by $d_G(x)$. It follows that the number of nodes of $T(H)$ that are examined by procedure Mark is also bounded by $d_G(x)$. Clearly, every node examined by Mark(x) is processed in constant time.

In the remainder of this paper a node w of $T(H)$ will be referred to as *marked* only if w remains marked at the end of procedure Mark (i.e.. w is marked but not subsequently unmarked). For a node w in $T(H), T(w)$ will denote the subtree of $T(H)$ rooted at w. For later reference, we make note of the following simple observations.

OBSERVATION 6. Let w be a marked node in $T(H)$. There must exist a child w' of w such that all the leaves in $T(w')$ are adjacent to x.

If this were not true, the node w could not possibly be marked.

OBSERVATION 7. Let w be a never marked or a marked, but not an unmarked, node of $T(H)$. There must exist a descendant w'' of w in $T(w)$ such that all the leaves in $T(w'')$ are nonadjacent to x.

If this were not true, w would have been marked and subsequently unmarked.

Let w be an arbitrary node of $T(H)$ and let $I(w)$ stand for the set of children of w which have a marked (and not subsequently unmarked) descendant in $T(H)$. Let $T'(w)$ stand for the subtree of $T(w)$ defined by

$$T'(w) = T(w) - \bigcup_{u \in I(w)} T(u).$$

Partition the leaves of $T'(w)$ into nonempty, disjoint sets $A(w)$ and $B(w)$, in such a way that x is adjacent to all the leaves in $A(w)$ and nonadjacent to all the leaves in $B(w)$.

OBSERVATION 8. w is the lowest common ancestor of any leaves a in $A(w)$ and b in $B(w)$.

To see this, note that every descendant w' of w in $T'(w)$ is such that $T(w')$ contains leaves from $A(w)$ or $B(w)$ only, for otherwise with u standing for a counterexample with the lowest level in $T'(w)$, u must be marked, a contradiction.

If $T(H)$ contains marked nodes, then the marked node with the lowest level in $T(H)$, denoted $\alpha(x)$ (or simply α, if no confusion is possible) plays a distinguished role in our algorithm. (If several marked nodes are at the same level, pick one arbitrarily.) Let

$$(7) \qquad (P) \quad w_1(=R),\ w_2,\ \cdots,\ w_p = \alpha(x) \qquad (p \geq 1)$$

stand for the unique path in $T(H)$ joining R and α. The path (P) is referred to as *complete* if no marked vertex in $T(H)$ lies outside (P).

For nodes w_j with $1 \leq j \leq p-1$ of a complete path (P), the subtree $T(w_j) - T(w_{j+1})$ contains no marked node; as before, we let

- $A(w_j)$ stand for the set of leaves in $T(w_j) - T(w_{j+1})$ that are adjacent to x;

• $B(w_j)$ stand for the set of leaves in $T(w_j) - T(w_{j+1})$ that are not adjacent to x.
For $w_p = \alpha(x)$, denote by

• $A(w_p)$ the set of all the leaves in $T(w_p)$ that are adjacent to x;

• $B(w_p)$ the set of all the remaining leaves in $T(w_p)$.

OBSERVATION 9. No w_k $(1 \leq k \leq p)$ on the path (P) is marked and subsequently unmarked.

This is obvious, by definition, for α; let j be the largest subscript for which w_j is a counterexample. However, since w_j was both marked and unmarked, it follows, in particular, that so was w_{j+1}, a contradiction.

OBSERVATION 10. Let w be an arbitrary unmarked node, or an improperly marked 1-node in (P). There exists a nonempty set S of leaves of $T(w)$, such that x is nonadjacent to all the leaves in S.

By Observation 9, any unmarked node in (P) cannot have been both marked and then, unmarked. Now the conclusion follows instantly from Observation 7.

OBSERVATION 11. If $d(R) = 1$ and R is marked, then R is properly marked.

This follows easily from the marking scheme: if $d(R) = 1$ and R is marked, but not unmarked then, trivially, $md(R) = 0$ and so $d(R) = md(R) + 1$, implying that R is properly marked.

Call a node w_j $(1 \leq j \leq p-1)$ of (P) *regular* if w_j is either a properly marked 1-node or else an unmarked 0-node. Otherwise, w_j is termed *special*. The path (P) is said to be *admissible* if the following conditions are satisfied.

(a1) (P) is complete;

(a2) there is at most one subscript k $(1 \leq k \leq p-1)$ such that the node w_k is special. Furthermore, if a special node exists, then the following conditions must be true

(a2.1) $k = p-2$ or $k = p-1$;

(a2.2) If $k = p-1$ then $|A(w_p)| = |B(w_p)| = 1$; furthermore,
$|B(w_k)| = 1$ whenever w_p is a 0-node and
$|A(w_k)| = 1$ whenever w_p is a 1-node.

(a2.3) If $k = p-2$ then
$|B(w_p)| = |A(w_{p-1})| = |A(w_k)| = 1$ and $B(w_{p-1}) = \emptyset$ whenever w_p is a 0-node;
$|A(w_p)| = |B(w_{p-1})| = |B(w_k)| = 1$ and $A(w_{p-1}) = \emptyset$ whenever w_p is a 1-node.

Note that, if $T(H)$ contains no marked nodes, then the path (P) is, trivially, empty and hence vacuously admissible. Now in our notation, Theorem 1 in Corneil et al. [4] can be formulated as follows.

PROPOSITION 3.1. *If H is a cograph, then $H + x$ is a cograph if and only if the path in $T(H)$ joining the root and $\alpha(x)$ is admissible and contains no special nodes.*

We are now ready to state a result which provides the theoretical basis for our algorithm to test condition (3). We assume the existence of an underlying graph $G = (V, E)$ which is in the process of being investigated by the recognition algorithm.

THEOREM 3.2. *If H is a P_4-sparse graph, then* (3) *is satisfied if and only if the path joining the root of $T(H)$ and $\alpha(x)$ is admissible.*

Proof. Let $H = (V_H, E_H)$ be described by the tuple $(T(H), SK(H))$, where $T(H)$ is the cotree associated with the canonical cograph $C(H)$. Assume that the vertex x is neutral with respect to H. If $T(H)$ contains no marked nodes, or if the path in $T(H)$ joining R and $\alpha(x)$ is admissible and contains no special nodes, then by Proposition 3.1, $C(H) + x$ is a cograph, and hence, no P_4 in $H + x$ contains x.

Now we may assume that the path (P) joining R and $\alpha(x)$ is admissible and contains a special node. In this case, a straightforward argument, which is left to the reader, shows that x is contained in precisely one P_4 in $C(H) + x$.

Conversely, assume that the (2) and (3) are satisfied. In particular, x must be neutral with respect to H. We only need prove that in the presence of special nodes, the path (P) joining R and $\alpha(x)$ is admissible.

Our proof relies on the following intermediate results that we present as facts. Write (P) as in (7).

FACT 1. (P) is complete.

Proof. Suppose not; let γ stand for the lowest marked node in $T(H)$ that does not belong to (P), and let w_i stand for the lowest common ancestor of α and γ in $T(H)$. We claim that

(8) γ and w_i are either both 0-nodes or both 1-nodes.

Suppose not; symmetry allows us to assume that γ is a 0-node and that w_i is a 1-node. By Observations 6–8 combined, we find leaves a, b, c, d in $A(\alpha), B(\alpha), A(\gamma), B(\gamma)$, respectively. Since w_i is a 1-node, we have $ad, bc, bd \in E$. Since γ is a 0-node, $cd \notin E$. It follows that $xcbd$ is a P_4. Now α must be a 1-node, for otherwise $ab \notin E$ and so $xadb$ would be a second P_4 containing x. Clearly, our choice of α implies that α and w_i are distinct nodes of $T(H)$. Since they are both 1-nodes, we find a 0-node w_j on the path from α to w_i. Let t be an arbitrary leaf in $T(w_j) - T(w_{j+1})$. Clearly, $ta, tb \notin E$ and $tc, td \in E$. We must have $xt \in E$, or else $xctd$ is a P_4, implying that x belongs to two distinct P_4's. However, now $txab$ is a P_4, contradicting that (3) is satisfied.

Now (8) allows us to assume without loss of generality that

both γ and w_i are 0-nodes.

(the case where both γ and w_i are 1-nodes is perfectly symmetric).

As before, by Observations 6–8 combined we find leaves a, b, c, d in $A(\alpha), B(\alpha), A(\gamma), B(\gamma)$, respectively. Since, by assumption, w_i and γ are distinct 0-nodes, we find a 1-node θ on the path in $T(H)$ from γ to w_i. Let t be an arbitrary leaf in $T(\theta) - T(c(\theta))$ (here, $c(\theta)$ is the child of θ that lies on the path from γ to θ). Obviously, we have $at, bt \notin E$ and $tc, td \in E$.

Observe that $xt \in E$, for otherwise $\{a, x, c, t, d\}$ induces two distinct P_4's. Now $axtd$ is a P_4 in $H + x$. Furthermore, note that α must be a 0-node, else $baxt$ is a second P_4 containing x. Since α and w_i are distinct 0-nodes, we find a 1-node w_j on the path from α to w_i. Let t' be a leaf in $T(w_j) - T(w_{j+1})$. We have $t'a, t'b \in E,\ t'c, t'd \notin E$. But now either $bt'xc$ or $xat'b$ is a P_4 depending on whether or not $xt' \in E$. Either case leads to a contradiction, and the proof of Fact 1 is complete. □

FACT 2. If w_k $(1 \leq k < p)$ is an improperly marked 1-node or an unmarked 1-node, then $k = p-1$ or $k = p-2$, depending on whether or not $\alpha(x)$ is a 0-node. Furthermore, if $k = p - 2$, then w_{p-1} is unmarked.

Proof. To begin, we claim that

(9) $$d(w_k)\ \geq 2.$$

This follows from the definition of the cotree, combined with Observation 11.

Clearly, (9) implies that $T(w_k) - T(w_{k+1}) \neq \emptyset$. By Observation 9, w_{k+1} cannot be marked and subsequently unmarked, and so $d(w_k) \geq md(w_k) + 1$. Since this inequality must, in fact, be strict, there exists a leaf d in $T(w_k) - T(w_{k+1})$ with $dx \notin E$.

We let, as usual, a, b stand for arbitrary leaves in $A(\alpha),\ B(\alpha)$, respectively. Obviously, since w_k is a 1-node, we have $ad, bd \in E$. It is easy to see that

if the statement is false, then we find distinct subscripts i, j
$(k < i < j \leq p)$ such that both w_i and w_j are 0-nodes.

To justify this observation, note that in case w_p is a 0-node we set $j \leftarrow p$ and since $k < p-1$ we set $i \leftarrow p-2$; in case w_p is a 1-node, we set $j \leftarrow p-1$ and $i \leftarrow k+1$.

Let t be an arbitrary leaf in $T(w_i) - T(w_{i+1})$. Since w_i is a 0-node, $ta, tb \notin E$; since w_k is a 1-node, $ad, bd, td \in E$. If w_p is a 0-node, then $xadb$ is a P_4; in addition, either $bdtx$ or $xadt$ is a P_4 depending on whether or not $xt \in E$.

We shall, therefore, assume that w_p is a 1-node. Let t' be an arbitrary leaf in $T(w_j) - T(w_{j+1})$. Note that $tt', t'a, t'b \notin E$ and $t'd \in E$; since w_p is a 1-node we have $ab \in E$. But now, $\{a, b, x, t, t', d\}$ induces at least two distinct P_4's containing x. To see that this is the case, note that $xt, xt' \notin E$, else $baxz, xzdb$ are P_4's with $z = t$ or $z = t'$ such that $xz \in E$. However, now $xadt$ and $xadt'$ are distinct P_4's containing x, contrary to our assumption.

Finally, we claim that

if $k = p - 2$, then w_{p-1} is unmarked.

If w_{p-1} were marked, then by Observation 6 we would find a vertex c in $T(w_{p-1}) - T(w_p)$ with $xc \in E$. Since w_{p-1} is a 0-node $ac, bc \notin E$; since w_k is a 1-node, $cd \in E$. But now, the set $\{a, b, c, d, x\}$ induces two distinct P_4's containing x (namely, $baxc$ and $bdcx$), a contradiction.

This completes the proof of Fact 2. $\square$

FACT 3. If w_k $(1 \leq k < p)$ is a marked 0-node, then $k = p-1$ or $k = p-2$, depending on whether or not $\alpha(x)$ is a 1-node. Furthermore, if $k = p-2$, then w_{p-1} is properly marked.

Proof. Trivially, $w_k \neq R$. By Observations 6 and 9 combined, there exists a leaf c in $T(w_k) - T(w_{k+1})$ such that $xc \in E$. We claim that

if the statement is false, then we find distinct subscripts i, j
$(k < i < j \leq p)$ such that both w_i and w_j are 1-nodes.

To justify this observation, note that if w_p is a 0-node, then since $k < p-2$ we set $j \leftarrow p-1$ and $i \leftarrow k+1$; if w_p is a 1-node, then set $j \leftarrow p$ and since $k < p-1$, we set $i \leftarrow k+1$.

As usual, we let a, b stand for arbitrary leaves in $A(\alpha)$, $B(\alpha)$, respectively. Let t be an arbitrary leaf in $T(w_i) - T(w_{i+1})$. Since w_i is a 1-node, we have $at, bt \in E$; since w_k is a 0-node, we have $ac, bc, tc \notin E$. We note that

(10) $\qquad w_p$ is a 0-node.

Otherwise, $baxc$ is a P_4, and either $taxc$ or $btxc$ is a P_4 depending on whether or not $xt \notin E$.

By (10), $j \neq p$; let t' be an arbitrary leaf in $T(w_j) - T(w_{j+1})$. Note that since w_p is a 0-node $ab \notin E$; now $xt, xt' \in E$, or else $\{b, z, a, x, c\}$ induces two distinct P_4's with $z = t$ or $z = t'$. But now, $btxc, bt'xc$ are distinct P_4's containing x, contrary to our assumption.

Finally, we claim that

if $k = p - 2$, then w_{p-1} is properly marked.

Suppose not; now by Observation 10, $T(w_{p-1}) - T(w_p)$ contains a leaf d with $xd \notin E$. However, with c as above, $\{c, x, a, d, b\}$ induces two distinct P_4's in $C(H) + x$, a contradiction.

This completes the proof of Fact 3. □

FACT 4. The path (P) contains at most one special node.

Proof. To begin, note that by Facts 2 and 3 combined, (P) cannot contain two distinct special nodes of the same kind (both 0-nodes or both 1-nodes). We let w_k $(1 \leq k < p)$ be a special 1-node and w_r $(1 < r < p)$ be a special 0-node in (P).

If w_p is a 0-node, then by Fact 2, $k = p - 1$. By Fact 3, $r = p - 2$ and w_{p-1} must be properly marked, a contradiction.

Thus w_p must be a 1-node. By Fact 3, $r = p-1$. By Fact 2, $k = p-2$ and w_{p-1} must be unmarked, a contradiction. □

To complete the proof of Theorem 3.2, we need only prove that the conditions (a2.2) and (a2.3) in the definition of the admissible path are satisfied. For this purpose, we distinguish between the following two cases.

Case 1. w_p is a 0-node.

If w_k is an improperly marked 1-node or an unmarked 1-node, then, by Fact 2, $k = p - 1$. For every choice of a leaf a in $A(w_p)$, b in $B(w_p)$, and d in $B(w_k)$, $xadb$ is a P_4. It follows that

$$|A(w_p)| = |B(w_p)| = |B(w_k)| = 1.$$

Furthermore, if w_k is a marked 0-node, Fact 3 implies that $k = p - 2$ and that w_{p-1} is properly marked. This implies that $B(w_{p-1}) = \emptyset$. Now for every choice of b in $B(w_p)$, t in $A(w_{p-1})$, and c in $A(w_k)$ we have $btxc$ a P_4. It follows that

$$|B(w_p)| = |A(w_{p-1})| = |A(w_k)| = 1$$

Case 2. w_p is a 1-node.

If w_k is a marked 0-node then, by Fact 3, we have $k = p - 1$. Note that for every choice of a in $A(w_p)$, b in $B(w_p)$, and c in $A(w_k)$, $baxc$ is a P_4, implying that

$$|A(w_p)| = |B(w_p)| = |A(w_k)| = 1$$

If w_k is an improperly marked 1-node or an unmarked 1-node then, by Fact 2, $k = p - 2$ and w_{p-1} is unmarked. By Observation 6, $A(w_{p-1}) = \emptyset$. Furthermore, for every choice of a in $A(w_p)$, d in $B(w_k)$ and t in $B(w_{p-1})$, $xadt$ is a P_4, and so

$$|A(w_p)| = |B(w_{p-1})| = |B(w_k)| = 1,$$

and the proof is complete. □

COROLLARY 3.3. *If* $|M_0 \cup M_1| > 2$, *then* (3) *is not satisfied.*

Proof. If $c_0 + c_1 > 2$ then the path (P) joining α and R cannot be admissible. The conclusion follows by Theorem 3.2. □

Two nodes of $T(H)$ play a distinguished role in Steps 2.2–2.3. First, $\alpha(x)$ stands, as before, for a marked node in $T(H)$ with the lowest level (ties being broken arbitrarily); next, $\gamma(x)$ is a candidate for a special node on the path joining α and R. (We shall write, simply, α and γ instead of $\alpha(x)$ and $\gamma(x)$ since no confusion is possible.) Step 2.2.2 is implemented by the procedure Find whose details are given below.

Procedure Find;
{returns a node that plays the role of α.}
1. **begin** Find $\leftarrow$ undefined;
2. **if** $c_0 + c_1 + c_2 = 0$ **then** Find $\leftarrow \Lambda$;
   {now there exist marked nodes in $T(H)$}

```
3.          case c0 + c1 of
4.             0:  if p(p(z)) is an unmarked node of T(H) for some z in M2 then
5.                   Find ← z
6.                 else begin
7.                   let z be a node in M2 such that z ≠ p(p(z')) for all z' ∈ M2;
8.                   Find ← z
9.                 end;
10.            1:  begin
11.                  let z be the unique node in M0 ∪ M1;
12.                  if z = p(z') or z = p(p(z')) for some z' ∈ M2 then
13.                     Find ← z'
14.                  else
15.                     Find ← z
16.                  end;
17.            2:  if for distinct z, z' in M0 ∪ M1, z' = p(z) or z' = p(p(z)) then
18.                  Find ← z
19.         endcase
20.   end; {Find}
```

OBSERVATION 12. Procedure Find runs in $d_G(x)$ time.

To see this, note that by a previous observation $|M_2|$ is bounded by $d_G(x)$. For each element in M_2 each of the tests in lines 4, 7, and 12 takes a constant time once a bit-vector representation for M_2 is assumed.

The following result shows that (3) is satisfied only if the node returned by the procedure Find can play the role of α. More precisely, we have Fact 5.

FACT 5. Let z be the node returned by the function Find. (3) is satisfied only if the following statements are satisfied:

(i) $z = \Lambda$ whenever $T(H)$ contains no marked nodes;

(ii) z and α coincide whenever $T(H)$ contains marked nodes.

Proof. To begin, note that procedure Find returns "undefined" whenever $c_0+c_1 > 2$. By Corollary 3.3, (3) is not satisfied. Next, line 2 in procedure Find guarantees that $z = \Lambda$ whenever $c_0 + c_1 + c_2 = 0$. To show that (ii) must also be satisfied, we shall rely on the following simple observations.

OBSERVATION 13. Let z be a marked node in $T(H)$ such that the parent of z or the grandparent of z (but not both) is either a marked 0-node or an improperly marked 1-node or an unmarked 1-node. Then (3) is satisfied only if z and α coincide.

By assumption, $T(H)$ contains marked nodes. As usual, let α stand for a marked node of the lowest level in $T(H)$. If (3) is satisfied, then by Theorem 3.2, the path (P) in $T(H)$ joining α and R is admissible. Thus z must belong to (P). But now, z must coincide with α, for otherwise we contradict Fact 2 or Fact 3 in the proof of Theorem 3.2.

OBSERVATION 14. Let $M_2 \neq \emptyset$ and $M_0 \cup M_1 = \emptyset$; (3) is satisfied only if α is either a node in M_2 whose grandparent is unmarked or, failing this, a node in M_2 that is grandparent of no node in M_2.

Clearly, $\alpha \in M_2$; let z be a node in M_2 such that $p(p(z))$ is unmarked. By Observation 13, z and α must coincide; now we may assume that no such node z exists. It follows that α is a node in M_2 that is grandparent of no node in M_2, as claimed.

OBSERVATION 15. Let $|M_0 \cup M_1| = 1$; (3) is satisfied only if α is either a node in M_2 whose parent or grandparent is the unique node in $M_0 \cup M_1$ or, failing this, α is the

unique node in $M_0 \cup M_1$.

Let t stand for the unique node in $M_0 \cup M_1$. If for some node z in M_2, $t = p(z)$ or $t = p(p(z))$, then by Observation 13, z and α coincide. If no such z exists in M_2, then $\alpha \notin M_2$ or else we contradict Fact 2 or Fact 3. It follows that α and t coincide.

OBSERVATION 16. Let $|M_0 \cup M_1| = 2$; (3) is satisfied only if α belongs to $M_0 \cup M_1$.

If $\alpha \notin M_0 \cup M_1$, then the path in $T(H)$ joining α and R cannot be admissible.

Now the conclusion follows immediately from Observations 12–16, and the proof of Fact 5 is complete. □

We assume that whenever the "unmarked w" statement is executed during Steps 2.2 and 2.3 with $w \in M_i$, the following statements are implicitly performed:

$$M_i \leftarrow M_i - \{w\};$$
$$c_i \leftarrow c_i - 1;$$
$$md(w) \leftarrow 0.$$

Step 2.2.3 is implemented by the procedure TestAdmissible whose details are spelled out next. As justified by Fact 5, we may use α for the node returned by procedure Find.

Procedure TestAdmissible;
{tests the path in $T(H)$ joining α and R for admissibility.}

1. **begin**
2. **if** α = undefined **then** return("no");
3. $\gamma \leftarrow \alpha$; **if** $\alpha = \Lambda$ **then exit**;
4. **if** ($p(\alpha) \in M_0$) or label($p(\alpha)$) = 1 and $p(\alpha) \notin M_2$ **then** $\gamma \leftarrow p(\alpha)$
5. **else if** ($p(p(\alpha)) \in M_0$) or (label($p(p(\alpha))$) = 1 and $p(p(\alpha)) \notin M_2$) **then** $\gamma \leftarrow p(p(\alpha))$;
   {to begin, check the path between γ and R}
6. $z \leftarrow \gamma$;
7. **if** label(z) = 0 **then**
8. $z \leftarrow p(z)$
9. **else** $z \leftarrow p(p(z))$;
10. **while** $z \in T(H)$ **do begin**
11. **if** $z \notin M_2$ **then** return("no") **else** unmark z;
12. $z \leftarrow p(p(z))$
13. **end**;
    {check whether an appropriate number of nodes remain marked}
14. **if** ($\gamma = p(p(\alpha))$) and (label(α) = 0) **then** {we know that $p(\alpha) \in M_2$}
15. unmark $p(\alpha)$;
16. **if** ($c_0 + c_1 + c_2 > 2$) or (($c_0 + c_1 + c_2 > 1$) and (γ not marked)) **then** return("no");
    {finally, check the conditions 2.2 and 2.3}
17. **case** γ **of**
18. $p(\alpha)$: **begin**
19. **if** label(α) = 0 **then**
20. **if** $|A(\alpha)| \neq 1$ or $|B(\alpha)| \neq 1$ or $|B(\gamma)| \neq 1$ **then** return("no")
21. **else begin** $b \leftarrow |B(\gamma)|$; $a \leftarrow |B(\alpha)|$; $c \leftarrow |A(\alpha)|$; $d \leftarrow x$ **end**
22. **else** {label(α) = 1}
23. **if** $|A(\alpha)| \neq 1$ or $|B(\alpha)| \neq 1$ or $|A(\gamma)| \neq 1$ **then** return("no")

```
24.                   else begin d ← |A(γ)|; b ← |A(α)|; a ← |B(α)|; c ← x end
25.            end;
26.        p(p(α)):   begin
27.               if label(α) = 0 then
28.                  if|B(α)| ≠ 1 or |A(p(α))| ≠ 1 or |A(γ)| ≠ 1 or B(p(α)) ≠ ∅
                        then
                     return("no")
29.                  else begin d ← |A(α)|; b ← |A(p(α))|; a ← |B(α)|; c ← x
                        end
30.               else {label(α) = 1}
31.                  if|A(α)| ≠ 1 or |B(p(α))| ≠ 1 or |B(γ)| ≠ 1 or A(p(α)) ≠ ∅
                        then
                     return("no")
32.                  else begin b ← |B(γ)|; a ← |B(p(α))|; c ← |A(α)|; d ← x
                        end
33.            end
34.       endcase
35.   end; {TestAdmissible}
```

OBSERVATION 17. Procedure TestAdmissible runs in $d_G(x)$ time.

To justify this claim, observe that the loop in lines 10–13 is executed at most $O(|M_2|)$ times and that each iteration takes constant time once a bit-vector representation for M_2 is assumed. The conditions (a2.2) and (a2.3) take $d_G(x)$ time to check if we are careful to keep the children of each node in $T(H)$ in a bit-vector form.

OBSERVATION 18. If γ is chosen in lines 4 and 5 of TestAdmissible, then exactly one of the following conditions hold true:

(i) label(α) = 0 (1), label(γ) = 1 (0), and $\gamma = p(\alpha)$;

(ii) label(α) = 0 (1), label(γ) = 0 (1), and $\gamma = p(p(\alpha))$; furthermore,

$p(\alpha)$ is properly marked whenever label(α) = 0,

$p(\alpha)$ is unmarked whenever label(α) = 1.

Observation 18 follows trivially from the code for lines 4 and 5.

OBSERVATION 19. If γ is chosen in lines 4 and 5 of TestAdmissible, then $\{a, b, c, d\}$ induces a P_4 with edges ab, bc, cd. Further, label(γ)=1 if and only if x is endpoint of a P_4.

Observation 19 follows from Observation 18 and the code for lines 23, 24 and 31, 32.

FACT 6. The path (P) in $T(H)$ from α to R is admissible if and only if the statement return("no") is not executed in TestAdmissible.

Proof. To begin, assume that (P) is admissible. If (P) contains no special node then α and γ coincide and we are done. We may therefore assume that (P) contains a special node. By Observation 18, this special node will be correctly determined in lines 4 and 5.

Next, lines 6–15 will unmark all properly marked 1-nodes in (P) from γ (exclusive) to R (inclusive) and, in case both α and γ are 0-nodes, $p(\alpha)$ which by Fact 3 is a 1-node is also unmarked. It follows that when line 16 is reached $c_0 + c_1 + c_2 \leq 2$, with equality if α and γ are both marked. Finally, it is easy to see that the admissibility of (P) implies that return("no") will not be executed in lines 17–34.

Conversely, assume that the statement return("no") is not executed in TestAdmissible. We only need prove that if $\alpha \in T(H)$, then the path (P) is admissible.

First, we claim that

$$(P) \text{ is complete.}$$

Otherwise, line 16 would have detected the presence of a marked node in $T(H)$ outside (P).

Next, we claim that

$$(P) \text{ contains at most one special node.}$$

Otherwise we would have executed the return("no") statement in line 11 in case (P) contained an unmarked node or an improperly marked node other than (possibly) γ, or line 16 if (P) contained a marked 0-node other then γ.

Finally, it is easy to see that since no return("no") was executed in lines 17–34, the conditions (a2.2) and (a2.3) are satisfied, and the path (P) is admissible. □

We note that by virtue of Facts 5 and 6, Theorem 3.2 can be reformulated as follows.

THEOREM 3.4. *If H is a P_4-sparse graph, then* (3) *is satisfied if and only if the statement return("no") is not executed in Steps* 2.1 *and* 2.2 *of Recognize.*

4. Algorithms II: Updating the data structures. In this section, we shall show how the data structures are updated once we have determined that $H+x$ is a P_4-sparse graph. We begin by introducing a result describing the canonical cograph of $H + x$ in terms of the canonical cograph of H.

THEOREM 4.1. *If $H+x$ is a P_4-sparse graph, then the following statements are satisfied:*

(i) *If x belongs to no P_4 in $C(H) + x$, then $C(H + x) = C(H) + x$.*

(ii) *If x is endpoint of a P_4 in $C(H) + x$, then $C(H + x) = C(H)$.*

(iii) *If x is midpoint of a P_4 in $C(H) + x$, then $C(H + x) = C(H) - y + x$, where $x = f(y)$.*

Proof. To show that (i) is satisfied, suppose that a set B containing x induces a P_4 in $H+x$. Obviously, B must contain some vertex a that is not in $C(H)$. By the definition of $C(H)$, a must be the endpoint of some P_4 induced by $A = \{a, b, c, d\}$ with edges ab, bc, cd with b, c, and d belonging to $C(H)$. Since x is neutral with respect to H, no more than two vertices from A can be in B. If $d \notin B$, then $B-\{a\}\cup\{d\}$ would induce a P_4 contradicting the assumption that x belongs to no P_4 in $C(H) + x$. Thus, $A \cap B = \{a, d\}$. Since all vertices outside A are neutral with respect to A, a and d must have identical adjacencies in $B - A$, contradicting that B induces a P_4.

Furthermore, (ii) follows from Proposition 2.4. To show that (iii) holds, we first note that y is endpoint of a P_4 in $C(H) + x$. Thus, with Greedy run as before up to $C(H) + x$. Since, by Theorem 2.3, x is endpoint of no P_4 in $C(H) + x$, x must belong to $C(H + x)$. □

Step 2.3 shall be further refined as follows.

Step 2.3 $\{H \leftarrow H + x$; update $T(H)$ and $SK(H)\}$
 if x belongs to no P_4's in $H + x$ **then**
  update $T(H)$
 else begin
  update $SK(H)$;
  if x is a midpoint **then**
   swap x with its image under the bijection f;
 end

Two circumstances would require an update of $T(H)$. If x belongs to no P_4 in $H+x$, then we need only add x to $T(H)$. To do this, we use Update1, a procedure similar

to that used by Corneil, Perl, and Stewart [4] to update their cotree. If x is midpoint of a P_4 in $C(H) + x$, then we need to remove the vertex that plays the role of y in (iii) of Theorem 4.1 and then incorporate x into the remaining cotree. This is done by procedure Update2.

The procedure Update performs any required update of $SK(H)$, determines which type of update to $T(H)$ is needed, and invokes the appropriate procedure. The variables a, b, c, d returned by TestAdmissible provide the information needed to test the conditions in Step 2.3 of Recognize. If $a = 0$, then x belongs to no P_4 in $H + x$. Otherwise, either c or d will contain x, depending on whether x is midpoint or endpoint respectively. If this P_4 contains no vertices already in $SK(H)$, then a new index in SK must be created. Otherwise, precisely two vertices from $\{a, b, c, d\}$ are in $SK(H)$. The two remaining vertices must be added to produce $SK(H + x)$. If x is a midpoint, then the other vertex not in $SK(H)$ plays the role of y in (iii) of Theorem 4.1.

Procedure Update;
{Main update procedure to execute Step 2.3}

```
1.  begin
2.     if a = 0 then Update1; {x is contained in no P4 in H + x}
3.     else begin
4.        if SKindex(a) = 0 then s ← a else s ← d;
5.        if SKindex(b) = 0 then k ← b else k ← c;
6.        i ← max{SKindex(a), SKindex(b)};
7.        if k = b or s = a then SKadj(i) ← SKadj(i) + 1;
8.        if SKsize(i) = 2 and SKadj(i) > 1 then begin
9.           swap the two elements of K_i in SK(i);
10.           add (s, k) to SK(i);
11.           SKindex(s) ← i; SKindex(k) ← i;
12.           SKtype(s) ← "S"; SKtype(k) ← "K";
13.           SKsize(i) ← SKsize(i) + 1
14.           end
15.        else begin
16.           q ← q + 1;
17.           add (a, b), (d, c) to SK(q);
18.           SKindex(a) ←; SKindex(b) ← q; SKindex(c) ← q; SKindex(d) ←
q;
19.           SKtype(a) ← "S"; SKtype(d) ← "S";
20.           SKtype(b) ← "K"; SKtype(c) ← "K";
21.           SKsize(q) ← 2; SKadj(q) ← 1;
22.           end
23.        if x = c then begin
24.           Update2;
25.           if s = d then begin
26.              Tree(d) ← Tree(a); Tree(a) ← 0;
27.              Swap (Tree(b), Tree(c))
28.           end
29.        end
30.     end
31.  end; {Update}
```

We shall now justify our construction of $SK(H)$. A function g is said to be *valid*

for a P_4-sparse graph H whenever g satisfies the requirements (sp1) and (sp2) of the function f in the definition of a special partition. The following theorem completes our justification of the procedure Update.

THEOREM 4.2. *If $H + x$ is P_4-sparse and $SK(H)$ contains a special partition Σ of H, then, at the conclusion of procedure Update, $SK(H + x)$ contains a special partition Σ' of $H + x$.*

Proof. Let $\Sigma = \{S_1, S_2, \cdots, S_q\}$ $(q \geq 1)$ be a special partition of H and let

$$g : \bigcup_{i=1}^{q} S_i \rightarrow V - \bigcup_{i=1}^{q} S_i$$

be a valid function for H.

If $a = b = c = d = 0$, then x belongs to no P_4 in $H + x$ and no update of SK is needed. If x belongs to precisely one P_4 in $H + x$, then no vertex on that P_4 belongs to any P_4 in H. Thus, with $\Sigma' = \Sigma \cup \{a, d\}$ as the special partition of $H + x$, the extension g' of g to $\cup_{i=1}^{q} S_i \cup \{a, d\}$ such that $g'(a) = b$, $g'(d) = c$ is valid for $H + x$.

If x belongs to more than one P_4 in $H + x$, then clearly, two vertices of the P_4 induced by $\{a, b, c, d\}$ must already be in SK (this follows from the structure of SK). Let i be the $SKindex$ of those vertices. Let (s, k) be the remaining endpoint and midpoint, respectively. If $SKadj(i)$ changes from 1 to 2 in line 7, then we must reorganize the ordered pairs in $SK(i)$ in accordance with our definition of g. This situation can occur only once for each index i, and only two pairs of vertices need to be changed. Once g is updated for the vertices already in $SK(i)$, we set $g(s) = k$ to obtain a function valid for $H + x$. □

The procedures for updating $T(H)$ are given next.

Procedure Update1;
    $\{x$ is contained in no P_4 in $H + x\}$
1.   **begin**
2.   **if** $\alpha = \Lambda$ **then**
3.   **if** all nodes in $T(H)$ were marked and subsequently unmarked **then**
4.   add x as a child of R
5.   **else** {no node in $T(H)$ was marked}
6.   **if** $d(R) = 1$ **then**
7.   make x a child of the (only) child of R
8.   **else begin**
9.   make the old root and x children of a new 0-node θ;
10.   make θ the only son of the new root
11.   **end**
12.   **else** {now α is the only marked node in $T(H)$}
13.   **if** label(α) $= 0(1)$ **then**
14.   **if** $md(\alpha) = 1(d(\alpha) - md(\alpha) = 1)$ **then begin**
15. $\lambda \leftarrow$ unique marked and unmarked (never marked) child of α in $T(\alpha)$;
16.   **if** λ is a leaf in $T(H)$ **then begin**
17.   make λ, x children of a new node θ;
18.   make θ a child of α
19.   **end**
20.   **else**
21.   make x a child of λ

```
22.                 end
23.             else begin {now md(α) ≠ 1 (d(α) − md(α) ≠ 1)}
24.                 add every marked child of α to a new node θ with
                     label(θ) = label(α);
25.                 if label(α) = 0 then begin
26.                     make x, θ children of a new node θ';
27.                     make θ' a child of α
28.                     end
29.                 else begin
30.                     make θ a child of p(α);
31.                     make x, α children of a new node θ';
32.                     make θ' a child of θ
33.                     end
34.             end
35.   end; {Update1}
```

Procedure Update2;

{swap x with nonadjacent endpoint a to produce $T(H+x)$}

```
1.   begin
2.       remove a from T(H);
3.       caselabel(α) of
4.           1:  if B(γ) = ∅ then begin
5.                   add b as a child of γ;
6.                   add x as a child of p(γ);
7.                   remove α from T(H)
8.                   end
9.               else begin
10.                  make b, d children of a new 0-node θ;
11.                  make θ, x children of α
12.              end;
13.          0:  begin
14.                  if B(γ) = ∅ then
15.                      add x as a child of p(γ)
16.              else begin
17.                  make p(α) and d children of a new 0-node θ;
18.                  make θ and x children of a new 1-node θ';
19.                  make θ' a child of γ;
20.                  end;
21.              if md(α) = 1 then begin
22.                  let α' be the marked and subsequently unmarked child of α;
23.                  if α' is a leaf in T(H) then
24.                      make α' a child of p(α)
25.                  else begin
26.                      make every child of α' a child of p(α);
27.                      remove α' from T(H)
28.                      end;
29.                  remove α from T(H)
30.              end;
31.              unmark α, unless already removed;
32.              if γ is marked then unmark γ
```

33. **end**
34. **endcase**
35. **end**; {Update2}

We will conclude this section with a discussion of the timing of the recognition algorithm. By Observations 5, 12, and 17 combined,

(11) *Step* 2.2 *of Stage* 2 *is performed in time bounded by* $d_G(x)$.

Next, we claim that

(12) *Procedure Update*1 *runs in* $d_G(x)$ *time.*

To see that this is the case, observe that all the transformations in Update1 can be carried out in constant time, except for line 24 which involves $O(A(\alpha))$ operations, which is bounded by $d_G(x)$, as claimed.

Further, we claim that

(13) *Procedure Update*2 *runs in* $d_G(x)$ *time.*

To see that this is the case, we note that by Observation 19, in case γ is a 1-node, x is an endpoint of a P_4 in $H+x$ and so lines 2–5 take a constant time to execute. Furthermore, if γ is a 0-node, then all the tree transformations entailed by removing a from $T(H)$ and adding x to $T(H)-a$ take a constant time with the exception of line 32 which requires $O(A(\alpha))$ time.

Finally, we claim that

(14) *Procedure Update runs in* $d_G(x)$ *time.*

To see that this is the case, observe that Update has no loops and the only procedures invoked are Update1 and Update2. Now the statement follows from (12) and (13).

We conclude our timing argument by noting that Stage 1 of the algorithm obviously takes constant time since only two vertices are processed. Now, (6), (11), and (14) combined can be summarized in the following theorem.

THEOREM 4.3. *Given a* P_4*-sparse graph* H *described by* $(T(H), SK(H))$ *and a given vertex* $x \notin H$*, the algorithm Recognize performs in time* $O(d_G(x))$ *one of the following:*

(i) *either determines that* $H+x$ *is not a* P_4*-sparse graph, or*

(ii) *incorporates* x *into* H*, updating* $T(H)$ *and* $SK(H)$ *accordingly.*

5. Tree representation for P_4**-sparse graphs.** Let G be a P_4-sparse graph represented by the tuple $(T(G), SK(G))$. We now address the problem of efficiently constructing the ps-tree representation of G. For this purpose we shall use the fact that $T(G)$ is the canonical cotree of G (i.e., the cotree corresponding to the canonical cograph $C(G)$ of G). We shall enumerate the vertices of $SK(i)$ as $s_1(i), \cdots, s_t(i)$ and $k_1(i), \cdots, k_t(i)$ where $t = SKsize(i)$ and $g(s_j) = k_j$ for $1 \leq j \leq t$. Whenever possible, we simply write s_j and k_j instead of $s_j(i)$ and $k_i(i)$.

We note that $s_1(i)$ and all vertices in K_i belong to $T(G)$, while $s_2, \cdots, s_t$ do not. We further note that if G is a spider, with partition $S \cup K \cup R$, then for any $x, y \in S \cup K$, $SKindex(x) = SKindex(y)$. For convenience, we shall refer to this index as $SKind(G)$, and define $s(G) = s_1(SKind(G))$, $sd(G) = SKsize(SKind(G))$. Our arguments make use of the following result.

THEOREM 5.1. *For each i, $(1 \leq i \leq q)$, there exist a unique 0-node $\lambda(i)$ and a 1-node $\lambda'(i)$ in $T(G)$ such that, setting $z = s_1(i)$, for any $v, w \in K_i$ with $zv \notin E$, $zw \in E$, the following are satisfied:*

$$\lambda(i) = p(z); \ \lambda'(i) = p(w); \ \lambda'(i) = p(\lambda(i)). \tag{15}$$

Furthermore,

$$\textit{either } \lambda(i) = p(v) \quad \textit{or} \quad \lambda''(i) = p(v) \quad \textit{with } \lambda(i) = p(\lambda''(i)). \tag{16}$$

Proof. Clearly, it it sufficient to show that if the statement is true for some induced subgraph H of G, then it is also true after incorporating x into H. We may assume that x is contained in some P_4 in $H + x$, for otherwise there is nothing to prove. We shall distinguish between the following two cases.

Case 1. x is endpoint of a P_4 in $H + x$.

By Theorem 4.1, $T(H+x) = T(H)$. The only new pair of vertices satisfying the criteria for $\{v, w\}$ is $\{f(z), f(x)\}$. Let $i = SKindex(x)$.

By Observation 19, γ is a 1-node in $T(H)$. If α is a 0-node, then the condition (a2.2) guarantees that

$$|A(\alpha)| = |B(\alpha)| = |B(\gamma)| = 1$$

Now, writing $A(\alpha) = \{v\}$, $B(\alpha) = \{z\}$, $B(\gamma) = \{w\}$, (15) is satisfied with α in place of λ and γ in place of λ'.

If α is a 1-node, then condition (a2.3) guarantees that

$$|A(\alpha)| = |B(p(\alpha))| = |B(\gamma)| = 1 \quad \text{and} \quad A(p(a)) = \emptyset.$$

Now, writing $A(\alpha) = \{v\}$, $B(p(\alpha)) = \{z\}$, $B(\gamma) = \{w\}$, (15) and (16) are satisfied with α in place of λ'', $p(\alpha)$ in place of λ, and γ in place of λ'.

Case 2. x is midpoint of a P_4 in $H + x$.

Now $\{f(z), x\}$ are the new candidates for $\{v, w\}$. Let u be the vertex such that $x = g(u)$. By Observation 19, γ must be a 0-node. First, if α is a 1-node, then condition (a2.2) translates as

$$|A(\alpha)| = |B(\alpha)| = |A(\gamma)| = 1.$$

Now write $A(\alpha) = \{v\}$, $B(\alpha) = \{u\}$, $A(\gamma) = \{z\}$. When lines 4–12 of Update2 are executed, condition (15) is verified with γ and $p(\gamma)$ in place of λ and λ' (in case $B(\gamma) = \emptyset$), or with θ in place of λ and α in place of λ' (in case $B(\gamma) \neq \emptyset$).

Finally, if α is a 0-node, then condition (a2.3) guarantees that

$$|B(\alpha)| = |A(p(\alpha))| = |A(\gamma)| = 1 \text{ and } B(p(\alpha)) = \emptyset.$$

Now, write $B(\alpha) = \{u\}$, $A(p(\alpha)) = \{v\}$, $A(\gamma) = \{z\}$. When lines 13–30 in Update2 are executed, conditions (15) and (16) hold true with either $p(\alpha)$ in place of λ'', γ in place of λ, and $p(\gamma)$ in place of λ', or else with $p(\alpha)$ standing for λ'', θ standing for λ and θ' standing for λ'.

To complete the proof of the theorem, note that conditions (a2.2) and (a2.3) guarantee, on the one hand, the uniqueness of $\lambda(i)$, and on the other, that no further alteration of $T(H + x)$ can separate λ and λ' or λ'', λ, and λ'. $\square$

Since for every i ($1 \leq i \leq q$), there is a *unique* $\lambda(i)$ with the properties mentioned in Theorem 4.2, we shall write simply λ, λ', λ'', dropping the reference to i.

To construct the tree representation of a P_4-sparse graph G, we need a way of incorporating the vertices of $SK(G)$ into the tree structure. For this purpose, a new type of node is needed; this is the 2-node which corresponds to the ② operation as in (1). A 2-node has precisely two children, one of them a 1-node and the other a single vertex or a 0-node. The details of this tree construction are spelled out in the following procedure.

```
Procedure Buildpstree(G);
   {Input: T(G), SK(G) for a P4-sparse graph G;
   Output: a ps-tree for G;}
1.   begin
2.      for i ← 1 to q do begin
3.         r ← SKsize(i);
4.         create a 2-node θ;
5.         create a 1-node θ';
6.         γ ← p(p(s1));
7.         make p(s1) a child of θ';
8.         if r = 2 then begin
9.            make s1 a child of θ;
10.           make k1 a child of θ';
11.        else begin
12.           create a 0-node α;
13.           make s2, ··· , sr children of α;
14.           make α a child of θ;
15.           if SKadj(i) = 1 then
16.              make k1 a child of θ';
17.           else
18.              make k2, ··· , kr children of θ';
19.        end
20.        make θ' a child of θ;
21.        if γ has any remaining children then
22.           make θ a child of γ;
23.        else begin
24.           make θ a child of p(γ);
25.           remove γ;
26.        end
27.     end
28.     if d(R) = 1 then R ← unique child of R;
29.  end; {Procedure Buildpstree}
```

The following result argues about the correctness and the running time of procedure Buildpstree.

THEOREM 5.2. *The tree $T1(G)$ returned by the procedure Buildpstree is precisely the ps-tree corresponding to G. Furthermore, $T1(G)$ is constructed in linear time.*

Proof. We only need prove that the root R of $T1(G)$ satisfies the following three conditions, as they extend easily to subtrees.

(t1) R is a 0-node of degree p whenever G is disconnected having p ($p \geq 2$) distinct components;

(t2) R is a 1-node of degree p whenever $\overline{G}$ is disconnected having p ($p \geq 2$) distinct

components;

(t3) R is a 2-node whenever G and $\overline{G}$ are connected.

Our proof relies of the following intermediate results.

FACT 7. If both G and $\overline{G}$ are connected, then the canonical cograph $C(G)$ induces a disconnected subgraph of $\overline{G}$ with precisely sd(G)+1 components, sd(G) of them containing single vertices.

Proof. Write $G = (V, E)$. By Proposition 2.1, G is a spider. $C(G)$ contains $\{s(G)\} \cup K$. Every vertex x in $N_G(s(G))$ is adjacent to every vertex in $V - \{x\}$, implying that $C(G)$ induces a disconnected subgraph of $\overline{G}$ with exactly sd(G) +1 components, sd(G) of them single vertices. □

FACT 8. Let $\overline{G}$ be disconnected; enumerate the components of $\overline{G}$ as $\overline{G_1}, \overline{G_2}, \cdots, \overline{G_p}$ with $p \geq 2$. Let I stand for the set of all the subscripts i $(1 \leq i \leq p)$ such that both G_i and $\overline{G_i}$ are connected. Let $r = \sum_{i=1}^{q} \overline{sd(G_i)}$. Then $\overline{C(G)}$ is disconnected and contains $p + r$ components, r of them being single vertices.

Proof. For every $i \in I$, Fact 7 guarantees that $\overline{C(G_i)}$ is disconnected and contains sd$(\overline{G_i})$+1 components, sd$(\overline{G_i})$ of which are single vertices. The conclusion follows. □

Observe that (t1) is implied by the following result.

FACT 9. If G is disconnected, then the tree $T1(G)$ is rooted at a 0-node whose degree equals the number of components of G.

Proof. Enumerate the components of G as $G_1, G_2, \cdots, G_p$ with $p \geq 2$. Trivially, the canonical cograph $C(G)$ is also disconnected with components $C(G_1), C(G_2), \cdots, C(G_p)$. Hence the canonical cotree $T(G)$ is rooted at a 1-node R' with $d(R') = 1$.

Let w stand for the unique child of R' in $T(G)$. By the previous argument w has precisely p children, corresponding to the components of $C(G)$. Since $d(R') = 1$, it follows that w cannot play the role of $\lambda(i)$ for $1 \leq i \leq q$. Consequently, when we exit the for loop (lines 2–28), w is left unchanged. Now line 29 guarantees that $T1(G)$ is rooted at w, as claimed. □

Next, note that (t2) is implied by the following result.

FACT 10. If $\overline{G}$ is disconnected, then the tree $T1(G)$ is rooted at a 1-node whose degree equals the number of components of $\overline{G}$.

Proof. Enumerate the components of $\overline{G}$ as $G_1, G_2, \cdots, G_p$ with $p \geq 2$. Let I stand for the set of all the subscripts i $(1 \leq i \leq p)$ such that both G_i and $\overline{G_i}$ are connected. With r as in Fact 8, the canonical cograph $C(G)$ induces a disconnected subgraph of $\overline{G}$, containing $p + r$ components, with r of them being single vertices. Consequently, the canonical cotree $T(G)$ is rooted at a 1-node R with $d(R) = p + r$. G is a spider with $S \cup K$ in $SK(G)$.

To see that R is the root of $T1(G)$, note that whenever $R = \lambda(i)$ for some $1 \leq i \leq q$, lines 22 and 24 in Buildpstree guarantee that the newly created 2-node becomes a child of R. By Proposition 2.1, G_i is a spider for every $i \in I$. Further, for each $i \in I$, $\lambda(i) = R$.

To see that R has degree p in $T1(G)$ note that the r children of R that are leaves in $T(G)$ are precisely the vertices which will be incorporated into 2-nodes in steps 16–19 of Buildpstree, leaving R with exactly p children, as claimed. □

Further, the following result implies (t3).

FACT 11. If both G and $\overline{G}$ are connected, then the tree $T1(G)$ is rooted at a 2-node.

Proof. Write $G = (V, E)$. Since both G and $\overline{G}$ are connected, G is a spider. Let s be the vertex from S in $C(G)$. $T(G)$ is rooted at a 1-node R of degree sd(G)+1. sd(G) of the children of R are single vertices, and the remaining one is a 0-node τ.

Since $\lambda'(SKind(G)) = R$, all single leaf children of R will be moved in lines 16–18 of Buildpstree. Thus the test in line 22 will fail and R will be replaced by the 2-node. □

Finally, to address the complexity of procedure Buildpstree, we note that each vertex in $SK(G)$ is moved at most once. Thus the procedure runs in time proportional to $|V|$. This completes the proof of Theorem 5.2. □

6. Conclusions and open problems. A graph G is P_4-sparse if no set of five vertices in G induces more than one chordless path of length three. P_4-sparse graphs find applications to network technology, group-based cooperation, cluster analysis, scheduling, and resource allocation, where graphs featuring "local density" properties are relevant. In these applications it is typical to equate local density with the absence of chordless paths of length three. Note that from this standpoint, the cographs [3] and P_4-reducible graphs [6] correspond, respectively, to the local density metrics described below:

(μ 1) the graph contains no induced P_4;

(μ 2) every vertex of the graph belongs to at most one induced P_4. Clearly, the P_4-sparse graphs correspond to the metric:

(μ 3) every set of five vertices contains at most one P_4.

In practical applications, metric (μ3) is less restrictive and more realistic than both (μ1) and (μ2).

In this work we have proposed several new characterizations of P_4-sparse graphs (see Theorems 2.3 and 2.5) and showed that they can be used, in conjunction with the tree representation, for the purpose of recognizing the P_4-sparse graphs in linear (and thus, optimal) time.

A number of problems remain open, however. It would be of interest to know whether the techniques developed in this paper, along with the data structures returned by the recognition algorithm, can be used to produce efficient solutions to other computational problems important in applications such as finding a maximum clique, a maximum stable set, an optimal coloring, clustering, minimum fill-in, minimum weight dominating set, hamiltonicity, and others.

Acknowledgment. The authors wish to express their gratitude to the referees for their exceptionally thorough review of the paper. The second author thanks Jingyuan Zhang for his help with LaTeX.

REFERENCES

[1] J. A. Bondy, U. S. R. Murty, *Graph Theory with Applications*, North-Holland, Amsterdam, 1976.

[2] D. G. Corneil and D. G. Kirkpatrick, *Families of recursively defined perfect graphs*, Congr. Numer., 39 (1983), pp. 237–246.

[3] D. G. Corneil, H. Lerchs, and L. Stewart Burlingham, *Complement Reducible Graphs*, Discrete Appl. Math., 3 (1981), pp. 163–174.

[4] D. G. Corneil, Y. Perl, and L. K. Stewart, *A linear recognition algorithm for cographs*, SIAM J. Comput., 14 (1985), pp. 926–934.

[5] C. Hoáng, Ph.D. thesis, McGill University, Montreal, Canada, 1985.

[6] B. Jamison and S. Olariu, *P_4-reducible graphs, a class of uniquely tree representable graphs*, Stud. Appl. Math., 81 (1989), pp. 79–87.

[7] ———, *A linear-time recognition algorithm for P_4-reducible graphs*, Proc. of the 9th Conference on Found. Software Technology and Theoretical Computer Science, Bangalore, India, 1989; also in Lecture Notes in Computer Science, 405, Springer-Verlag, New York, 1989, pp. 1–19.

[8] ———, *A tree representation for P_4-sparse graphs*, Discrete Appl. Math., 35 (1992), pp. 115–129.

[9] H. Lerchs, *On cliques and kernels*, Department of Computer Science, University of Toronto, Toronto, Ontario, March 1971.

[10] ———, *On the clique-kernel structure of graphs*, Department of Computer Science, University of Toronto, Toronto, Ontario, October, 1972.

SIAM J. COMPUT.
Vol. 21, No. 3, pp. 407-437, June 1992

© 1992 Society for Industrial and Applied Mathematics
001

ON THE EXACT COMPLEXITY OF STRING MATCHING: UPPER BOUNDS*

ZVI GALIL† AND RAFFAELE GIANCARLO‡

Abstract. It is shown that, for any pattern of length m and for any text of length n, it is possible to find all occurrences of the pattern in the text in overall linear time and at most $\frac{4}{3}n-\frac{1}{3}m$ character comparisons. In fact, the bound on the number of character comparisons is usually tighter than this, for the bound is expressed in terms of the structure of the pattern. The algorithm here need not have any knowledge of the alphabet. This improves the best previous bound of $1.5n-.5(m-1)$ obtained by Colussi [*Inform. and Comput.*, to appear] and Apostolico and Crochemore [Tech. Report TR89-75, LITP, Université de Paris, Paris, France, 1989]. In a companion paper [*SIAM J. Comput.*, 20 (1991), pp. 1008-1020], the authors show a lower bound for on-line algorithms that is equal to $\frac{4}{3}n-\frac{1}{3}m$ for $m=3$. For $m=1, 2$, n character comparisons is optimal. This algorithm is based on a new analysis of the string matching algorithm by Colussi. Moreover, this new analysis of Colussi's algorithm confirms the experimental results showing that his algorithm performs very well in practice [*Inform. and Comput.*, to appear].

Key words. string matching, string searching, text editing, computational complexity, worst case behavior

AMS(MOS) subject classifications. 68Q20, 68Q25, 68U15

1. Introduction. Given a computational problem, the ultimate goal is to determine its computational complexity *exactly*. However, this task is usually impossible for several reasons. One reason is that the exact complexity or even the constant factors depend on the machine model. Consequently, an exact determination of the time required by the problem is possible only if we restrict the model of computation, for example, by using a comparison tree model or considering straight line programs over a given set of operations.

For several problems involving order statistics the exact number of comparisons is known. The simplest of these are computing the maximum or minimum in $n-1$ comparisons (folklore), computing the maximum *and* the minimum in $\lceil\frac{3}{2}n-2\rceil$ comparisons [20] and computing the largest two elements in $n-1+\lceil\log n\rceil$ comparisons [16], [22]. For many other problems, such knowledge is partial or not available. For instance, exact bounds for selecting the third largest element are known for all but a finite set of cases [14], [15], [23]. For the classic problem of sorting, the exact number of comparisons is known only up to $n=12$ [1], [9] and for some special cases [17]. A fascinating open problem is to find the exact number of comparisons for determining the median: the best upper bound is $3n$ [21] and the best lower bound is $2n$ [4].

In this paper, we use a RAM with uniform cost criterion, or alternatively a binary decision tree model [1]. We evaluate the time complexity of algorithms counting *all* operations and the number of comparisons. For example, the straightforward algorithm for computing the minimum of n numbers takes $O(n)$ time and performs $n-1$ comparisons in our model.

---

* Received by the editors October 15, 1990; accepted for publication (in revised form) May 21, 1991.

† Department of Computer Science, Columbia University, New York 10027, and Tel-Aviv University, Tel-Aviv, Israel. The work of this author was supported in part by National Science Foundation grant CCR 88-14977.

‡ AT&T Bell Labs, Murray Hill, New Jersey 07974. This author is on leave from the University of Palermo, Italy. Part of this work was performed while this author was at Columbia University, while he was supported in part by National Science Foundation grant CCR 88-14977 and by the Italian Ministry of University and Scientific Research, Project "Algoritmi e Sistemi di Calcolo."

We investigate the exact complexity of string matching over a general alphabet. Let $w = w[1, s]$ be a string. We denote by $w[i, j]$ the substring of w that starts at position i and ends at position j of $w[1, s]$. We denote the positions $i, i+1, \cdots, j$ as $[i, j]$. *String Matching* is the problem of finding all occurrences of a given pattern $p[1, m]$ in a given text $t[1, n]$. We say that the pattern occurs at text position i if $t[i+1, i+m] = p[1, m]$. By general alphabet we refer to the case of an infinite alphabet or of a finite alphabet unknown to the algorithm (thus it must work if the alphabet is $\{a, b, c, d\}$ or $\{\heartsuit, \diamondsuit, \clubsuit, \spadesuit\}$). Most known linear-time, i.e., $O(n+m)$, string matching algorithms do not depend on the knowledge of the alphabet, since they only compare symbols. Another common feature of these algorithms is that they *preprocess* the pattern, i.e., they gather knowledge about the structure of the pattern and then start looking for it in the text. We count only the character comparisons that these algorithms perform while looking for occurrences of the pattern in the text. However, we do account for the time taken by the preprocessing. We establish new upper bounds on the number of character comparisons sufficient for an algorithm to correctly perform string matching. In a companion paper we provide lower bounds [11].

For quite some time, the best upper bound known has been $2n - m$ and corresponds to the number of character comparisons made by the Knuth, Morris, and Pratt algorithm [18] (KMP for short). Even the Boyer–Moore algorithm [5] (BM for short) could not beat the $2n - m$ bound. Indeed, Knuth [18] showed that BM performs at most $7n$ character comparisons, assuming that the pattern does not occur in the text. Under the same assumptions, Guibas and Odlyzko [13] reduced this bound to $4n$ and, very recently, Cole [6] showed that $3n$ is a tight bound (up to small order magnitude terms). Apostolico and Giancarlo [3] designed a variation of BM that achieves the $2n - m$ bound, irrespective of how many times the pattern occurs in the text. All of the analyses of BM mentioned here are quite intricate. Crochemore and Perrin [8] devised a new time-space-optimal string matching algorithm that performs at most $2n - m$ character comparisons. Recently, Colussi [7] devised an ingenious algorithm, which we refer to as SM, and showed that it performs at most $1.5n - .5(m-1)$ character comparisons (however, as we shall see, the true bound is much more involved). Independently, Apostolico and Crochemore [2] obtained a bound of $1.5n - m + 1$ by means of a simple modification of KMP. Their algorithm is a special case of Colussi's algorithm. Note that we can use the failure function used by KMP to derive deterministic finite automata [1] which perform string matching in exactly n comparisons. But this class of algorithms must know the alphabet to work correctly and have a running time that does depend (by a multiplicative factor) on the alphabet size. The algorithms we are interested in have a running time independent of the alphabet size.

We denote by $c(n, m)$ the maximal number of comparisons needed by a linear-time string matching algorithm, excluding any preprocessing step. It follows from the discussion above that $c(n, m) \leq 1.5n - .5(m-1)$. Let $c_{on\text{-}line}(n, m)$ be as $c(n, m)$ for on-line algorithms, that is, algorithms whose access to the text is limited to a sliding window of size m. Moreover, the window can be aligned with $t[i, i+m-1]$ if and only if the algorithm has already decided whether an occurrence of the pattern can start at position k of the text, for all k, $1 \leq k < i$. The best bounds currently known for $c_{on\text{-}line}(n, m)$ have been the same as the ones for $c(n, m)$.

We need a few definitions. A string $x[1, m]$ is c *periodic* if and only if $x[1, m-c] = x[c+1, m]$. We refer to c as a period of x. *The period* of a string x is the minimal integer l such that l is a period of x. We say that $x[1, m]$ is *strongly periodic* if and only if $m = kl + l'$, $k > 1$, $l' < l$, and l is the period of x. Given a pattern $p[1, m]$ whose period is z, $m = kz + z'$, $0 \leq z' < z$, the *periodic decomposition* of p is a sequence of integer

triples $(z_1, z'_1, k_1), \cdots, (z_s, z'_s, k_s)$ such that $(z_1, z'_1, k_1) = (z, z', k)$; $p[1, z_{i-1}+z'_{i-1}]$ is strongly periodic of period z_i (and thus $z_{i-1}+z'_{i-1} = k_i z_i + z'_i$), for $1 < i \leqq s$; $p[1, z_s + z'_s]$ is not strongly periodic. Such a decomposition can be computed in $O(m)$ time using the preprocessing algorithm in [18]. The results of this paper can be summarized as follows:

(a) A new analysis of the algorithm SM [7] showing that it performs at most $n + \lfloor (n-m)(z'_s/z_s + z'_s) \rfloor \leqq n + \lfloor n - m/2 \rfloor$ character comparisons. Our sharper bound confirms the experiments by Colussi showing that his algorithm performs very well in practice because for most patterns $s = 1$ and z'_1 is smaller than z_1.

(b) Based on our analysis of SM, we devise a new algorithm and show that it performs at most $n + \lfloor (n-m)(\min(\frac{1}{3}, (z'_s+2)/2(z_s+z'_s))) \rfloor \leqq \frac{4}{3}n - \frac{1}{3}m$ character comparisons. Therefore, $c(n, m) \leqq c_{on\text{-}line}(n, m) \leqq \frac{4}{3}n - \frac{1}{3}m$. In the companion paper [11], we show a lower bound for $c_{on\text{-}line}(n, m)$ that is equal to $\frac{4}{3}n - \frac{1}{3}m$ for $m = 3$.

We remark that it is possible to prove that SM performs at most $n + \lfloor (n-m) \times (\min(z, m-z)/m) \rfloor \leqq n + \lfloor n - m/2 \rfloor$ character comparisons and that our new algorithm performs at most $n + \lfloor (n-m)(\min(\frac{1}{3}, \min(z, m-z)+2/2m)) \rfloor$ character comparisons. The interested reader is referred to [12] for proofs. Here we give proofs for the weaker bounds in (a) and (b) in order to simplify the presentation.

The $O(n+m)$ time bound of both algorithms is independent of the alphabet size. Both algorithms make at most n character comparisons for the important class of nonperiodic patterns, i.e., $z = m$. We remark that all the known linear-time string matching algorithms require more that n comparisons for this class of patterns.

The analysis of the algorithms presented here is complicated by the fact that they are oblivious: They sometimes forget the result of some comparisons when the pattern is moved over the text. These comparisons may have to be repeated in the future. Obliviousness in a string matching algorithm is not new. Indeed, the difficulty of the analysis of BM given in [3], [6], [13], and [18] is mostly due to the obliviousness of BM. However, none of the techniques in [3], [6], [13], and [18] could be used in our case since the obliviousness of BM is different than that of our algorithms, as explained in § 3.

This paper is organized as follows. In § 2 we show that, in order to bound $c(n, m)$ for any n and m, we can restrict our attention to patterns that are not strongly periodic, i.e., patterns having period of size z and length $m = z + z'$, $0 \leqq z' < z$. Such a reduction simplifies the presentation of our analysis of SM as well as of its improved version. We review SM in § 3. Section 4 contains all the combinatorial results needed for the analysis which is presented in § 5. An improved algorithm is presented and analyzed in § 6.

2. A reduction. We show that, in order to bound $c(n, m)$, we need consider only patterns that are not strongly periodic. We recall the following well-known *periodicity lemma* [18], [19].

LEMMA 1. *Let $y[1, n]$ be a string. If k_1 and k_2 are periods of y and $k_1 + k_2 \leqq n + \gcd(k_1, k_2)$, then $\gcd(k_1, k_2)$ is also a period of y.*

Let $l_i = z_i + z'_i$ denote the length of $p[1, z_i + z'_i]$ in the periodic decomposition of $p[1, m]$, $1 \leqq i \leqq s$.

FACT 1. *Assume that $p[1, m]$ is strongly periodic and consider its periodic decomposition. For any fixed i, $1 < i \leqq s$, $p[1, k_i z_i]p[1, l_i] = (p[1, z_i])^{k_i+1} p[1, z'_i]$ cannot be prefix of $p[1, 2z_{i-1} + z'_{i-1}] = (p[1, z_{i-1}])^2 p[1, z'_{i-1}]$.*

Proof. Notice that in the periodic decomposition of $p[1, m]$, z_{j-1} cannot be a multiple of z_j, for any j, $2 \leqq j \leqq s$. Otherwise, by the periodicity lemma, z_{j-1} cannot be

the period of a strongly periodic string. Thus, $z'_{j-1} \neq z'_j$. Moreover, $z_j > z'_{j-1}$. Otherwise, $z_{j-1} + z_j \leqq l_{j-1}$ and, since z_{j-1} and z_j are both periods of $p[1, l_{j-1}]$, $\gcd(z_{j-1}, z_j)$ is also a period of that string by the periodicity lemma. But then, z_{j-1} cannot be the period of a strongly periodic string.

Assume that for some fixed i, $1 < i \leqq s$, $p[1, k_i z_i]p[1, l_i]$ is a prefix of $p[1, 2z_{i-1} + z'_{i-1}]$. The occurrence of $p[1, l_i]$ at $(k_i - 1)z_i + 1$, $z_{i-1} + 1$ and $k_i z_i + 1$ imply that $p[1, l_i]$ has periods $|l_i - z'_{i-1}|$ and $|z'_i - z'_{i-1}|(z_{i-1} + z'_{i-1} = k_i z_i + z'_i)$. Recall that it also has periods z_{i+1} and z_i and that $z'_i \neq z'_{i-1}$. Notice that $|l_i - z'_{i-1}| = l_i - z'_{i-1}$ since $z_i > z'_{i-1}$.

Assume that $z'_i - z'_{i-1} < 0$ $(z'_i - z'_{i-1} > 0$, respectively). Since $(l_i - z'_{i-1}) + (z'_{i-1} - z'_i) \leqq l_i$ $(z_i + (z'_i - z'_{i-1}) \leqq l_i$, respectively), $u_i \equiv \gcd(l_i - z'_{i-1}, z'_{i-1} - z'_i)$ $(q_i \equiv \gcd(z_i, z'_i - z'_{i-1})$, respectively) is a period of $p[1, l_i]$ be the periodicity lemma. In the first case $u_i = \gcd(z_i + z'_i - z'_{i-1}, z'_{i-1} - z'_i) \leqq z'_{i-1} - z'_i < z_i$ and u_i divides z_i. Also in the second case $q_i < z_i$ and q_i divides z_i. But then, z_i cannot be the period of a strongly periodic string. □

We now sketch an algorithm that, given all the occurrences of $p[1, l_s]$ in a text $t[1, n]$, will find all occurrences of $p[1, m]$ in $t[1, n]$. The algorithm has s stages and each stage has a list of candidates. During stage i, $i = s, s-1, \cdots, 2$, the ordered list of candidates satisfies the following invariant: all occurrences of $p[1, l_i]$ are in the list and each of those is a potential occurrence of the pattern in the text. This invariant is trivially satisfied for $i = s$ since $p[1, l_s]$ is a prefix of $p[1, m]$ and the algorithm knows all of its occurrences in $t[1, n]$. The last stage will produce all occurrences of p in t. We now describe stage i, $2 \leqq i \leqq s$.

Stage i. Consider the ordered list of candidates. By the periodicity lemma, the distance between two candidates is at least z_i. Divide the list into subsequences of maximal length such that the distance between any two candidates in the same subsequence is z_i. Let $q_1 < q_2 < \cdots < q_r$ be one of those subsequences. Consider q_b, an element of the chosen subsequence. Assume that $b > r - k_i + 1$. Notice that $p[1, l_{i-1}] = (p[1, z_i])^{k_i} p[1, z'_i]$ cannot occur at q_b; otherwise the maximality of the subsequence would be contradicted. Since $p[1, l_{i-1}]$ is a prefix of the pattern no occurrence of the pattern in the text can start at q_b for $r - k_i + 1 < b \leqq r$. Assume that $b < r - k_i + 1$. Notice that an occurrence of $(p[1, z_i])^{k_i+1} p[1, z'_i]$ starts at q_b. Thus, by Fact 1, $p[1, 2z_{i-1} + z'_{i-1}]$ cannot occur at q_b. Since $p[1, 2z_{i-1} + z'_{i-1}]$ is a prefix of the pattern and of $p[1, l_{i-1}]$, none of those two latter strings can occur at q_b, $1 \leqq b < r - k_i + 1$. If q_{r-ki+1} exists, it is the only candidate of the subsequence that is an occurrence of $p[1, l_{i-1}]$ and a potential occurrence of the pattern. Such position can be found in $O(r)$ time and only it survives for the next stage. Processing in a similar fashion the remaining subsequences of the list of candidates, we get a new list of candidates for the next stage that satisfies the invariant. Notice that the new list is smaller by at least a factor of $k_i \geqq 2$ and can be obtained in linear time.

Stage 1. We have the list $j_1, \cdots, j_r$ of potential occurrences of the pattern in the text. Moreover, it is also a list of all occurrences of $p[1, l_1]$. A position j in the list is an occurrence of the pattern in the text if and only if $j + vz_1$ is also in the list, for all v, $1 \leqq v < k$. This stage can be implemented in $O(r)$ time.

Since there can be at most $O(n)$ occurrences of $p[1, l_s]$ in the text and, at each stage, the list is processed in time linear in its size and the size is reduced by at least $\frac{1}{2}$, the total time of the algorithm is $O(n)$ and without any character comparisons. Thus we have the following lemma.

LEMMA 2. *Let $p[1, m]$, $m = kz + z'$, $k \geqq 1$, $z' < z$, be a pattern having period of size z and let $t[1, n]$ be a text. Let (z_s, z'_s, k_s) be the last term in the periodic decomposition of $p[1, m]$. For any algorithm $\mathscr{A}$ that finds all occurrences of $p[1, z_s + z'_s]$ in $t[1, n]$ in $f(n, z_s + z'_s)$ character comparisons there is an algorithm $\mathscr{A}'$ that finds all occurrences of*

$p[1, m]$ in $t[1, n]$ in the same number of character comparisons spending $O(n)$ additional time.

From now on, we consider only patterns of period size z and length $z+z'$, $0 \leqq z' < z$.

3. Colussi's algorithm. The algorithm that we present is a blend of the two best known and most efficient string matching algorithms devised so far: Knuth–Morris–Pratt [18] and Boyer–Moore [5]. It has been obtained by Colussi [7] using the correctness proof of programs as a tool to improve algorithms.

Given an alignment of the pattern $p[1, m]$ with text characters $t[i+1, i+m]$, KMP checks whether these two string are equal proceeding from left to right. BM performs the same check proceeding from right to left. Both algorithms then shift the pattern over the text by some precomputed amount and the check is repeated on the resulting new alignment. The KMP and BM approaches to string matching seem to be orthogonal and no efficient and mutually advantageous way of combining them had been known. The main difficulty in combining the two approaches is to find a partition of the set of m pattern positions such that there is an efficient strategy to shift the pattern over the text when a mismatch is found. Colussi [7] designed such partition and the corresponding strategy. The starting point of our presentation is KMP.

We need a few definitions. Given a string $w[1, m]$ and an index j, $1 \leqq j < m$, we say that $k\min(j)$ is defined and equal to d if d is the minimal integer such that $w[1, j]$ is d periodic and $w[j-d+1] \neq w[j+1]$. Thus, $k\min(j)$ is defined if a periodicity ends at j. If no such d exists we say that $k\min(j)$ is undefined. We refer to each position j of the pattern, $1 < j \leqq m$, having $k\min(j-1)$ defined as *nohole*. We refer to the remaining pattern positions as *holes*. Note that 1 is always a nohole since we have not defined $k\min(0)$.

Let the pattern be aligned with $t[i+1, i+m]$. Assume that KMP has found out that $p[1, j] = t[i+1, i+j]$ and that $p[j+1] \neq t[i+j+1]$ for some j, $0 \leqq j < m$. The pattern must be shifted over the text by some amount. There are two possibilities: shift past text position $i+j+1$ (where the mismatch occurred) and not shift past text position $i+j+1$ (thus having some overlap with the part that was matched). KMP chooses the first possibility when there is no integer g such that $p[1, j]$ is g periodic and $p[j+1] \neq p[j-g+1]$, i.e., $j+1$ is a hole, since it can be easily shown that there cannot be any occurrence of the pattern in the interval $[i+1, i+j+1]$. KMP chooses the second possibility when there is an integer g such that $p[1, j]$ is g periodic and $p[j+1] \neq p[j-g+1]$, i.e., $j+1$ is a nohole. Then, KMP shifts the pattern over the text by $k\min(j)$ positions since no occurrence of the pattern can be in the interval $[i+1, i+k\min(j)]$. Moreover, there may still be an occurrence of the pattern at $i+k\min(j)+1$, since $p[1, j-k\min(j)] = p[k\min(j)+1, j] = t[i+1+k\min(j), i+j]$ and a pattern character different from $p[j+1]$ will be aligned with $t[i+j+1]$ (recall the definition of $k\min$).

Colussi observed that if $j+1$ is a nohole and $p[j+1] \neq t[i+j+1]$, we need not know that $p[1, j] = t[i+1, i+j]$ to deduce that the pattern must be shifted by $k\min(j)$ positions over the text: Let the pattern be aligned with $t[i+1, i+m]$ and assume that all the noholes in $p[1, j]$ match the corresponding text positions in $t[i+1, i+j]$ and that $p[j+1] \neq t[i+j+1]$, $j+1$ a nohole. Then, it can be shown that there can be no occurrence of the pattern in the text in the interval $[i+1, i+k\min(j)]$. Moreover, we can still use some of the knowledge acquired about $t[i+1, i+j]$ after the shift by $k\min(j)$ takes place. Indeed, all the noholes in $p[1, j-k\min(j)]$ will match the corresponding text positions in $t[i+k\min(j)+1, i+j]$. Therefore, the string matching can be restarted by comparing the first nohole beyond $j-k\min(j)$ with the corresponding text position beyond $i+j$. Let $h_1 < h_2 < \cdots < h_{nd}$, $h_{nd} \leqq m-1$, be the pattern positions such that $k\min$ is defined and let first(x) denote the least integer y such that

$x \leqq h_y$. The following lemma is a precise statement of Colussi's observation [7] (see Fig. 1).

LEMMA 3. *Let the pattern be aligned with* $t[i+1, i+m]$. *Assume that* $p[h_s+1] = t[i+h_s+1]$ *for* $1 \leqq s < r \leqq nd$, *and* $p[h_r+1] \neq t[i+h_r+1]$. *Let* $i_{\text{new}} = i + k\text{min}(h_r)$. *Then, there is no occurrence of the pattern in the interval* $[i+1, i_{\text{new}}]$ *and the pattern can be shifted over the text by* $k\text{min}(h_r)$ *positions. Moreover,* $p[h_s+1] = t[i_{\text{new}}+h_s+1]$ *for* $1 \leqq s < \text{first}(h_r - k\text{min}(h_r))$, *and the algorithm can be restarted by testing whether* $p[h_{\text{first}(h_r - k\text{min}(h_r))}+1] = t[i_{\text{new}} + h_{\text{first}(h_r-k\text{min}(h_r))}+1]$.

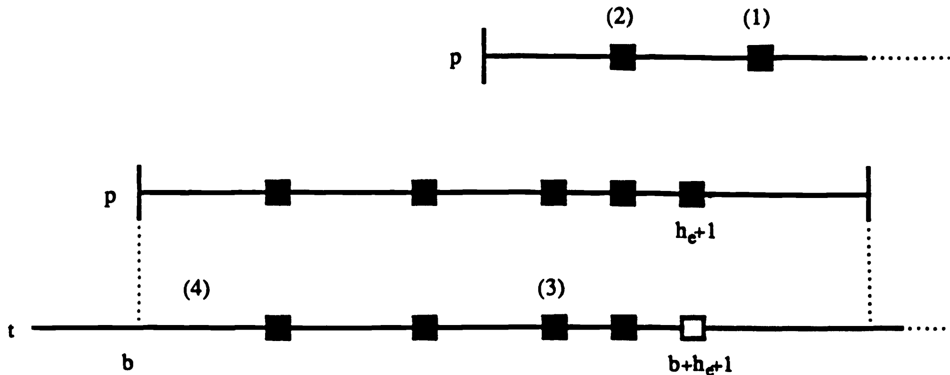


FIG. 1. *A mismatch with a selected position (a nohole) in iteration b. The shift is by the period* $k\text{min}(h_e)$ *of the interrupted periodicity. The algorithm resumes with the first nohole aligned with a text position not before the position of the mismatch* (1), *since smaller noholes (e.g.,* (2)) *must match. The comparison of* (3) *is forgotten, because after the shift it is aligned with a hole. On the other hand, if* (4) *was not compared before iteration b, it will not be compared at all.*

An operative conclusion that can be drawn from Lemma 3 is that when checking (from left to right) if $p[1, m] = t[i+1, i+m]$, we can compare those pattern positions $j+1$ having $k\text{min}(j)$ defined and shift by $k\text{min}(j)$ in case of a mismatch. Now, assume we know that all the noholes of the pattern match the corresponding text characters, i.e., $p[h_s+1] = t[i+h_s+1]$, $1 \leqq s \leqq nd$. We must decide in which order to compare the holes of the pattern with the corresponding text positions and what to do in the case that a mismatch is found. If we find out that $p[\hat{j}+1] \neq t[i+\hat{j}+1]$, $\hat{j}+1$ a hole, we can shift past text position $i+\hat{j}+1$. (Recall that a shift with overlap must have $k\text{min}(\hat{j})$ defined.) In order to guarantee the longest shifts while allowing the algorithm to retain more of the knowledge it acquired about $t[i+1, i+m]$, all the holes of the pattern are processed in decreasing order, that is, from right to left. Another advantage for this order is that by filling up the holes from right to left we completely match a suffix of the pattern.

Lemma 4 describes more precisely such processing (see Fig. 2). Let $h_{nd+1} > h_{nd+2} > \cdots > h_{m-1} > h_m = 0$ be all pattern positions such that $k\text{min}$ is undefined, $h_{nd+1} \leqq m-1$. Given a position j of the pattern, let $r\text{min}(j)$ be the minimal period of $p[1, m]$ greater than j.

LEMMA 4. *Let the pattern be aligned with* $t[i+1, i+m]$. *Assume that* $p[h_s+1] = t[i+h_s+1]$, $1 \leqq s < r$, *and that* $p[h_r+1] \neq t[i+h_r+1]$, $r > nd$. *Let* $i_{\text{new}} = i + r\text{min}(h_r)$. *Then, there is no occurrence of the pattern in the interval* $[i+1, i_{\text{new}}]$. *Moreover* $p[1, m - r\text{min}(h_r)] = t[i_{\text{new}}+1, i+m]$ *and the algorithm can be restarted by testing whether* $p[h_{\text{first}(m-r\text{min}(h_r))}+1] = t[i_{\text{new}} + h_{\text{first}(m-r\text{min}(h_r))}+1]$.

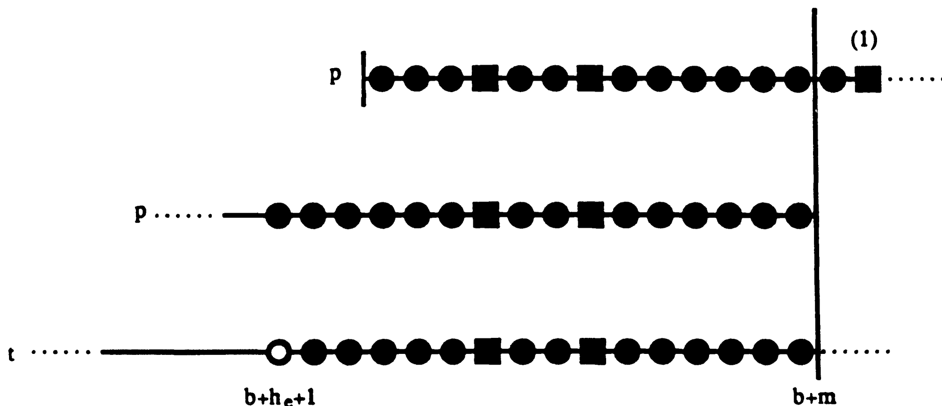


FIG. 2. *A mismatch with a hole. A complete suffix of the pattern is matched. The shift is by the smallest period of the pattern past the mismatch. (In the first alignment a suffix of the pattern is shown and after the shift a prefix is shown.) The algorithm resumes as in Fig.* 1. *After the shift the algorithm will only compare text positions greater than* $t_{\text{last}} = b + m$.

The operative conclusion that we can draw from Lemma 4 is that when we have completed a left to right pass and start a right to left pass, we are guaranteed that there is no need to consider text positions $t[i + r\min(h_r) + 1, i + m]$ every again, since a prefix of the pattern matches $t[i + r\min(h_r) + 1, i + m]$ and each future shift affects only the length of the match, since we always shift by a period of a prefix at least as large.

The subsequences $h_1 < h_2 < \cdots < h_{nd}$ and $h_{nd+1} > h_{nd+2} > \cdots > h_m$ provide a partition of the integers $\{0, \cdots, m-1\}$ which induces a partition on the pattern positions $\{1, \cdots, m\}$. The way to put these two sequences together to perform string matching is given by Lemmas 3 and 4. Indeed, we check whether $p[1, m] = t[i+1, i+m]$, using the sequence $h_1 + 1, h_2 + 1, \cdots, h_{nd} + 1, h_{nd+1} + 1, \cdots, h_m + 1$ in the order given. This sequence can be computed in $O(m)$ time and $2m - 1$ character comparisons using the preprocessing algorithm of KMP [18].

We define two tables disp and start having $m + 1$ entires each as follows: $\text{disp}[i] = k\min(h_i)$ and $\text{start}[i] = \text{first}(h_i - k\min(h_i))$, $1 \leqq i \leqq nd$; $\text{disp}[i] = r\min(h_i)$ and $\text{start}[i] = \text{first}(m - r\min(h_i))$, $nd < i \leqq m$; $\text{disp}[m+1] = z$ and $\text{start}[m+1] = \text{first}(m - z)$, where z is the period of the pattern. The last entry in these tables is used to restart the algorithm correctly when an occurrence of the pattern in the text is found. These tables are computed in the preprocessing stage.

In the pseudocode given below, variable *p*start is equal to the index from which the algorithm starts examining the sequence $h_1, \cdots, h_m$ at the beginning of each iteration. Variable *t*last denotes an index such that text characters in $t[1, t\text{last}]$ are not examined anymore by the algorithm. Variable b denotes an index such that the algorithm has already found all occurrences of the pattern in $t[1, b]$ while it has still to find the ones in $t[b+1, n]$. Variable e points to the element in the sequence $h_1, \cdots, h_m$ that is currently being used for comparison.

```
    Algorithm SM
begin
  b ← 0; pstart ← 1; tlast ← 0
  repeat
    e ← pstart;
```

```
      while e ≦ m and tlast < b + h_e + 1 and p[h_e + 1] = text [b + h_e + 1] do
   e ← e + 1;
        if e = m + 1 or tlast ≧ b + h_e + 1 then occurrence found
        if e > nd then tlast ← b + m;
        b ← b + disp [e];
        pstart ← start [e];
     until b > n − m;
   end
```

We say that an integer i is an iteration if Algorithm SM sets variable $b = i$ during its execution. An iteration i corresponds to an alignment of the pattern with $t[i+1, i+m]$.

We can observe a potentially serious limitation to a good performance of SM: After the pattern is shifted by some amount, there are situations in which the algorithm forgets that some of the text positions match the corresponding pattern positions and may compare them again. Consider a nohole $h_s + 1$ of the pattern, for some fixed $s \leqq nd$. It can happen that while $k\min(h_s)$ is defined, there may exist a nohole $h_{s'} + 1$, $s < s' \leqq nd$, such that $k\min(h_s - k\min(h_{s'}))$ is undefined. In other words, noholes are unstable under shifts corresponding to interrupted periodicities. The effect of this instability is that a text position matching a nohole in a given iteration may be aligned with a hole in the next iteration and be considered again if the algorithm starts matching pattern positions from right to left. Figure 1 gives an example of matched text positions that may be compared again.

Obliviousness in a string matching algorithm is not new: BM forgets part of what it knows about the text after a shift. This phenomenon has been extensively studied. However, none of the techniques used by KMP [18], Guibas and Odlyzko [13], Galil [10], Apostolico and Giancarlo [3], and Cole [6] to account for the obliviousness of BM could be used in our case. Intuitively, the obliviousness of BM is macroscopic whereas that of SM is microscopic. Indeed, BM forgets that contiguous chunks of text match some pattern substrings, whereas SM forgets that some text characters, not necessarily contiguous and irregularly distributed over the text, match some pattern characters.

Another property of SM is that it skips over some text positions while matching noholes. Then the pattern is shifted over the text. The text positions skipped over and to the left of the current alignment may be left untested, i.e., they are never compared with any pattern position. Figures 1 and 2 give a qualitative description of this phenomenon. Again, we remark that the text positions left untested are irregularly distributed over the text.

In order to obtain a tight analysis of SM, we would like to use all the untested text positions to pay for the mismatches and some of the repeated comparisons resulting from the obliviousness of SM. Achieving this goal requires tight upper bounds on the number of comparisons forgotten after a shift and a flexible and precise charging scheme that allows to amortize character comparisons against untested text positions. Such tasks are complicated by the arbitrariness and irregularity of the distributions (over the text) of forgotten comparisons and untested positions.

4. Some properties of strings. In order to analyze Algorithm SM and its improvement, we need to derive some properties of $k\min$ and to study some new combinatorial properties of strings. Throughout this section, let l be the maximal integer such that a string x has $x[1] = \cdots = x[l]$ and $x[l] \neq x[l+1]$.

FACT 2. *Let $x[1, h]$ be a k periodic prefix of x and assume that $h > k + l$. There can be no other period k' of $x[1, h]$ such that $|k - k'| \leqq l$.*

Proof. We give a proof only for the interval $[k+1, k+l]$. The proof for the other interval is similar. Assume that $x[1, h]$ is k' periodic, $k+1 \leqq k' \leqq k+l < h$. This implies that $x[k+l+1] = x[k+l+1-k'] = x[1]$. But, since $x[1, h]$ is also k periodic, $x[k+l+1] = x[l+1]$. Therefore, $x[1] = x[l+1]$, a contradiction to the definition of l. □

FACT 3. *If $j \geqq l+1$ and $k\min(j)$ is defined, then $k\min(j) > l$.*

Proof. Assume that, for $j \geqq l+1$, $k\min(j) \leqq l$. Since $k\min(j) \leqq l$ is a period of $x[1, j]$ we must have that $x[1, j] \in x[1]^*$, which is impossible since $x[l+1] \neq x[1]$ and $j \geqq l+1$. □

In what follows we derive relationships between the holes of a string and the periodicities of its substrings.

FACT 4. *Let $x[1, h]$ be a k periodic prefix of x. If $j - k$ is a nohole, $k < j \leqq h$, then j is a nohole.*

Proof. We assume that $k\min(j-1-k)$ is defined and show that $k\min(j-1)$ is defined. By definition of $k\min$, there exists an integer r such that $x[1, j-k-r-1] = x[r+1, j-k-1]$ and $x[j-k-r] \neq x[j-k]$. Using the k periodicity of $x[1, h]$, we obtain $x[1, j-k-r-1] = x[r+k+1, j-1]$ and $x[j-k-r] \neq x[j]$. Thus, $k\min(j-1)$ is defined. □

Given a k periodic prefix $x[1, h]$ of x, a hole $j - k$ may not carry over to j. The next fact establishes a relationship between "missing holes" and periodicity.

FACT 5. *Let $x[1, h]$ be a k periodic prefix of x. Assume that $j - k$ is a hole while j is not, for some j, $k < j \leqq h$. Then $i = k\min(j-1)$ satisfies $l < i < k$.*

Proof. Assume that $i \geqq k$ for the given j satisfying the hypothesis. If $i = k$, we have that $x[1, j-1-k] = x[k+1, j-1]$ and that $x[j] \neq x[j-k]$. This latter inequality contradicts the k periodicity of $x[1, h]$. If $i > k$, we note that because $i \leqq j-1, j-k-1 > 0$. We show that the k periodicity of $x[1, h]$ and the fact that $k\min(j-1) = i$ must imply that $k\min(j-1-k)$ is defined. This will contradict the assumption that $j - k$ is a hole: From the definition of $k\min$, $x[1, j-i-1] = x[i+1, j-1]$ and $x[j-i] \neq x[j]$. Combining these conditions with the facts that $i > k$ and $x[1, h]$ is k-periodic, we obtain that $x[i+1, j-1] = x[i+1-k, j-k-1] = x[1, j-i-1]$ and $x[j-k] \neq x[j-i]$, which imply that $k\min(j-k-1) \leqq i-k$, a contradiction. Thus, $i < k$. There is no $j \leqq l$ satisfying the hypothesis of the fact. Now, for $j \geqq l+1$, $i > l$ by Fact 3. □

Let v be a position of x such that $k\min(v-1)$ is defined. We say that v is *reachable* from position $u < v$ if there exists an increasing sequence of integers $u_1, u_2, \cdots, u_s$ such that $u_1 = u$, $u_s = v$, $k\min(u_1 - 1)$ is undefined and $k\min(u_{i+1} - 1) = u_i - 1$, for $1 \leqq i < s$. Notice that $u > 1$ and is the only hole in the sequence. We denote the sequence of integers $u = u_1, u_2, \cdots, u_s = v$ as $(u \Rightarrow v)$. We say that $(u_1 \Rightarrow v_1)$ and $(u_2 \Rightarrow v_2)$ are disjoint if they have no integers in common.

LEMMA 5. *Let $x[1, h]$ be a k periodic prefix of x. Assume that there is a position j, $k < j \leqq h$, such that $j - k$ is a hole whereas j is not. Then, there is a position q, $l+1 < q \leqq k$, from which j is reachable. Moreover, if two distinct positions j_1 and j_2 satisfy the assumptions of the lemma then $(q_1 \Rightarrow j_1)$ and $(q_2 \Rightarrow j_2)$ are disjoint.*

Proof. There is no $j \leqq l+1$ satisfying the hypothesis, since by definition of l and of $k\min$, no position j in $[1, l]$ can have $k\min(j-1)$ defined and $x[1, l+1]$ is not k periodic, for any $k < l+1$. Therefore, assume that $l+1 < j$.

Since j is nohole, $i = k\min(j-1)$ is defined. Since $j - k$ is a hole by hypothesis, we can apply Fact 5 to prefix $x[1, h]$ to obtain $l+1 < i+1 \leqq k$. If $k\min(i)$ is undefined, j is reachable from $i+1$, $l+1 < i+1 \leqq k$. Otherwise, we notice that $x[1, j-1]$ is i periodic and that $i+1$ is a nohole in this string whereas position 1 is a hole. Thus, we can now

apply Fact 5 to $x[1, j-1]$ with $j' = i+1$, $k' = i$, and $i' = k\min(i)$. Iterating the above reasoning, we get a decreasing sequence of indices that must end in a hole q, $l+1 < q \leqq i+1 \leqq k$.

Given q_1 and q_2 from which j_1 and j_2 are reachable, respectively, we now show that $(q_1 \Rightarrow j_1)$ and $(q_2 \Rightarrow j_2)$ are disjoint. The proof is by contradiction. Assume that such two sequences are not disjoint. We first notice that j_1 cannot be in $(q_2 \Rightarrow j_2)$, since $k\min(j_2 - 1) < k < j_1$. Similarly, j_2 cannot be in $(q_1 \Rightarrow j_1)$. Since the two sequences are not disjoint (by assumption), there must exist two indices f and g such that $u_f \in (q_1 \Rightarrow j_1)$, $u'_g \in (q_2 \Rightarrow j_2)$, $u_f = u'_g$ and $u_{f+1} \neq u'_{g+1}$. Without loss of generality, let $u'_{g+1} < u_{f+1}$. From the definition of reachability, we know that $x[1, u_{f+1} - 1]$ is $u_f - 1$ periodic, with $x[u_{f+1}] \neq x[u_{f+1} - u_f + 1]$. Analogously, we know that $x[1, u'_{g+1} - 1]$ is $u_f - 1$ periodic, with $x[u'_{g+1}] \neq x[u'_{g+1} - u_f + 1]$. This latter inequality contradicts the $u_f - 1$ periodicity of $x[1, u_{f+1} - 1]$.

Thus, $(q_1 \Rightarrow j_1)$ and $(q_2 \Rightarrow j_2)$ are disjoint and $q_1 \neq q_2$. □

LEMMA 6. *Let $x[1, m]$ be a string, z be its period, $m = z + z'$, $0 \leqq z' < z$. Let l be the maximal integer such that x has a prefix equal to $x[1]^l$ and let $x[1, h]$ be a k periodic prefix of x, $k < h$. Choose an integer $b \leqq h$. Assume that $j_i > \max(k, b)$, $1 \leqq i \leqq s$, are all the noholes in $x[1, h]$ such that $j_i - k$ is a hole and j_i is reachable from a hole in $x[1, \min(k, b)]$.*

If $b \leqq k$,

$$s \leqq \max\left(0, \left\lfloor \frac{b-1}{2} \right\rfloor\right). \tag{1}$$

Otherwise,

$$s \leqq \max\left(0, \left\lfloor \frac{k-2}{2} \right\rfloor\right). \tag{2}$$

Proof. For each j_i satisfying the hypothesis of the lemma, let $q_i \leqq \min(k, b)$ denote a hole from which j_i is reachable. By Lemma 5, each q_i is distinct and greater than $l+1$. From the inequality $q_i > l+1$, we can immediately deduce that $s = 0$ when $\min(k, b) \leqq l+1$. In what follows, we consider the case $\min(k, b) > l+1$.

For each $(q_i \Rightarrow j_i)$, let u_i be the minimal integer in that sequence greater than b. By Lemma 5, all $(q_i \Rightarrow j_i)$ are disjoint. Thus all u_i's are distinct and the correspondence between q_i's and u_i's is one to one. Let $d_i = k\min(u_i - 1) + 1$. Note that all d_i's are distinct ($d_i \in (q_i \Rightarrow j_i)$). We obtain the bounds by bounding the number of d_i's. In a one-to-one fashion, we associate with the d_i's a set of integers r_i in the interval I containing the d_i's. The two sets are disjoint and therefore their size is bounded by half the size of their union or half the number of the elements of I that are "used."

We must have $l+1 < d_i \leqq \min(k, b)$. Indeed, $l+1 \leqq q_i - 1 \leqq k\min(u_i - 1) = d_i - 1 \leqq k\min(j_i - 1)$, and $k\min(j_i - 1) < k$ by Fact 5. Using the definition of reachability and the fact that u_i is the least integer in $(q_i \Rightarrow j_i)$ larger than b, $k\min(u_i - 1) = d_i - 1 < b$.

We divide the d_i's in two sets B and C defined as follows: $B = \{d_i\colon d_i \leqq \min(b, k) - l\}$; $C = \{d_i\colon b - l < d_i \leqq \min(b, k)\}$.

It is obvious that B and C partition the d_i's when $b \leqq k$. We prove that we have a partition also in the case that $k < b$. The proof is by contradiction. Assume that there exists a d_g such that $d_g \notin B \cup C$. Then, $k - l < d_g \leqq \min(k, b - l)$. Since $x[1, h]$ is k periodic, $x[1, u_g - 1]$ is $d_g - 1$ periodic, $k < b$ and $u_g > b$, $x[1, b]$ is both k periodic and $d_g - 1$ periodic. Since $b > d_g + l - 1$, we apply Fact 2 to $x[1, b]$ (with $k = d_g - 1$), to obtain that no period of $x[1, b]$ is in $[d_g, d_g + l - 1]$. But k is in that interval. A contradiction.

We associate each $d \in B$ with $d+l$. Obviously, the correspondence is one to one. Moreover, $d+l \neq d_i$ for $1 \leqq i \leqq s$, since by the $(d_i - 1)$ and $d-1$ periodicity of $x[1, b]$ we have $x[l+1] = x[d+l] \neq x[1] = x[d_i]$. We also have $l+1 < d+l \leqq \min(k, b)$. We can now obtain the claimed bounds by assigning a suitable number of positions in $[1, \min(k, b)]$ to elements in C that nave not been assigned to elements of B and that are different from any d_i, $1 \leqq i \leqq s$. We distinguish two cases.

If $k < b$ ($k \geqq b$, respectively), then $|C| \leqq l-1$ ($|C| \leqq l$, respectively). Noting that positions $[1, l+1]$ are different from any d_i and have not been assigned to elements in B, we can assign to elements in C a corresponding number of these positions to obtain bound (2) ((1), respectively). □

5. Analysis of Algorithm SM. We need a few definitions. An iteration r *matches* a suffix of the pattern if $t[r+h_i+1] = p[h_i+1]$, $1 \leqq i \leqq nd$, and, at least, $t[r+m] = p[m]$. Two iterations r and u, $r < u$, have a *suffix-prefix* overlap if r matches a suffix of the pattern and $u \leqq r+m-1$. For any iteration u, let $t\text{last}(u)$ be the value of $t\text{last}$ in algorithm SM when u starts.

LEMMA 7. *$u < t\text{last}(u)$ if and only if u has a suffix-prefix overlap with an iteration $u' < u$.*

Proof. Assume $u < t\text{last}(u)$. Let $u' < u$ be the last iteration that advanced t last. Note that $0 = t\text{ last}(0)$, so u cannot be the first iteration. We have $t\text{ last}(u) = u'+m$ and since $u < t\text{ last}(u)$, $u \leqq u'+m-1$. Therefore u has an overlap with u'. Moreover, $e > nd$ at the end of iteration u', otherwise t last could not be updated during that iteration. Since $\max(h_{nd}, h_{nd+1}) = m-1$, and $e > nd$ at the end of iteration u', we know that $p[m]$ has been compared with $t[u'+m]$ during that iteration and that u' cannot use $k$min as a shift function since all the noholes match. If u' did not match any suffix of the pattern, we must have $p[m] \neq t[u'+m]$. But, in that case, there would be a shift by m of the pattern over the text (recall the definition of $r$min and the fact that u' cannot use $k$min for shifts). Therefore, the iteration succeeding u' is $\hat{u} = u'+m > u$, a contradiction. Thus $p[m] = t[u'+m]$ and u has a suffix-prefix overlap with u'.

Assume that u has a suffix-prefix overlap with u', $u' < u$. We first show that u cannot have a suffix-prefix overlap with any other iteration $\hat{u} < u$. Assume the contrary. Thus, $u - \min(\hat{u}, u') \leqq m-1$. Since both $\hat{u}$ and u' matched a suffix of the pattern, there was a shift by at least z of the pattern over the text at the end of both iterations. Thus $\min(\hat{u}, u') + z \leqq \max(\hat{u}, u')$ and $\max(\hat{u}, u') + z \leqq u$ implying that $\min(\hat{u}, u') + 2z \leqq u$. A contradiction since $u - \min(\hat{u}, u') \leqq m-1 < 2z$ ($m = z + z'$, $z' < z$).

Since u has a suffix-prefix overlap with u', u' must have matched a suffix of the pattern and set $t\text{ last} = u'+m$. We claim that no iteration $\hat{u}$, $u' < \hat{u} < u$ can advance t last. Assume $\hat{u}$ advances t last and hence $e > nd$ at the end of iteration $\hat{u}$. Since $\max(h_{nd}, h_{nd+1}) = m-1$, and $e > nd$ at the end of iteration u', we obtain that $p[m]$ has been compared with $t[\hat{u}+m]$. If $p[m] \neq t[\hat{u}+m]$, the pattern would be shifted by at least m implying $u - u' \geqq m$. A contradiction, since u and u' overlap. If $p[m] = t[\hat{u}+m]$, then $\hat{u}$ matches a suffix of the pattern and u has a suffix-prefix overlap with $\hat{u}$ (recall $u' < \hat{u} < u$ and $u - u' \leqq m-1$). A contradiction to the fact that u has suffix-prefix overlap with u'. Therefore, when u starts $t\text{ last}(u) = t\text{ last} = u'+m$ and since $u - u' \leqq m-1$, $u < t\text{ last}(u)$. □

The iterations of Algorithm SM are divided into sequences of consecutive iterations. There are two kinds of sequences: *light* and *heavy*. Informally, a heavy sequence is a maximal sequence of iterations such that the first matches a suffix of the pattern and every other iteration has a suffix-prefix overlap with some iteration in the sequence preceding it. A heavy sequence can be followed by a heavy or light sequence. A

nonempty sequence of iterations between two consecutive heavy sequences is a light sequence. As will be explained later, light sequences are easier to analyze and pay for themselves.

Formally, a sequence $s_1, s_2, \cdots, s_y$ of consecutive iterations is *light* if

(1) s_1 does not match a suffix of the pattern and it is either the first iteration of the string matching algorithm or it is preceded by an iteration that ends a heavy sequence.

(2) $s_i \geqq t\,\mathrm{last}\,(s_i)$, $1 < i \leqq y$ and either s_y is the last iteration of the algorithm or u matches a suffix of the pattern, where u is the iteration succeeding s_y.

A sequence $u_1, u_2, \cdots, u_g$ of consecutive iterations is *heavy* if

(1) u_1 matches a suffix of the pattern and it is either the first iteration of the string matching algorithm or it is preceded by an iteration that ends a sequence, either light or heavy. Notice that in both cases u_1 advances t last to $u_1 + m$ since it matches a suffix of the pattern.

(2) $u_i < t\,\mathrm{last}\,(u_i)$, $1 < i \leqq g$ and either u_g is the last iteration of the algorithm or $s \geqq t\,\mathrm{last}\,(s)$, where s is the iteration succeeding u_g.

By Lemma 7, the preceding definition implies that s has no suffix-prefix overlap with any iteration preceding it. Moreover, it also implies that $u_i, 1 < i \leqq g$, has a suffix-prefix overlap with some u_j, $1 \leqq j < i$, as we show next. Our claim follows from Lemma 7 if u_1 is the first iteration of the algorithm. Assume u_1 is not the first iteration of the algorithm. By Lemma 7, u_i has a suffix-prefix overlap with some $u' < u_i$, $u_i - u' \leqq m - 1$. Assume also that $u' < u_1$. Since both u' and u_1 match a suffix of the pattern, the pattern is shifted over the text by at least z at the end of both iterations. Therefore, $u' + z \leqq u_1$ and $u_1 + z \leqq u_i$ implying $u_i - u' \geqq 2z > m = z + z'$, $z' < z$, a contradiction, since we assume that $u_i - u' \leqq m - 1$. Therefore $u' = u_j$ for some j, $1 \leqq j < i$.

In order to ensure that each iteration of the algorithm has a successor, we add a dummy iteration that follows the last iteration of the algorithm. Such dummy iteration does nothing, is not part of any light or heavy sequence, and is not regarded as the last iteration of the algorithm. We set it equal to n.

We will use a charging mechanism in which each position in the text may pay for at most one comparison. Given an iteration j we say that a position $d > j$ of the text is *free* if $t[d]$ has never been considered by the string matching algorithm or it has been compared with one or more characters of the pattern but, for each comparison, a different position d', $d' \leqq j$, of the text paid for it. A position $d \leqq j$ of the text is *busy* if it paid for exactly one comparison performed by the algorithm.

Given an iteration s, let start (s) be the value of e in algorithm SM at the beginning of iteration s. Given an iteration v starting either a light or a heavy sequence, let $c(v)$ be the number of noholes in the pattern between zero and $h_{\mathrm{start}(v)}$. Notice that $c(v) = \mathrm{start}\,(v) - 1$. For notational convenience, we define $c(v) = 0$ for the dummy iteration. In our account of the number of character comparisons performed by algorithm SM we maintain the following invariant for any iteration s starting a sequence.

INVARIANT 1. Start $(s) \leqq nd + 1$ *and all positions of the text in* $t[s+1, s+h_{\mathrm{start}\,(s)}]$ *aligned with holes of the pattern are free whereas those remaining match the corresponding positions of the pattern. All positions in* $t[s + h_{\mathrm{start}\,(s)} + 1, n]$ *are free and, in the worst case, all positions in* $t[1, s]$ *are busy.*

5.1. The analysis of light sequences. Let $s_1, s_2, \cdots, s_y$ be a light sequence and let u_1 be either the iteration that starts the following heavy sequence or the dummy iteration. We estimate the number of comparisons performed by the algorithm during iteration s_1 and we prove that, if s_2 is not the dummy iteration, it satisfies invariant 1

when it starts. Then, we inductively extend the same analysis to the remaining iterations in the sequence. We will charge each iteration in the sequence. This charge is related but not equal to the number of character comparisons done by the algorithm during that iteration. If iteration s makes r comparisons, we will charge it $r+c(s)-c(s')$ comparisons, where s' is the next iteration. Thus s pays for the $c(s)$ comparisons done before s, but does not pay for the $c(s')$ comparisons that s' is going to inherit. We show that s can pay for the number of comparisons being charged to it using only free text positions. The free text positions used to pay become busy at the end of iteration s. (This is what we meant by saying that light sequences pay for themselves. In fact, each item of a light sequence pays for itself.)

We assume that s_1 satisfies invariant 1 when it starts. Thus, $t[s_1+h_1+1]=p[h_1+1], \cdots, t[s_1+h_{\text{start}(s_1)-1}+1]=p[h_{\text{start}(s_1)-1}+1]$ at the beginning of s_1. This amounts to $c(s_1)$ matches that Algorithm SM has identified during previous iterations. None of these text characters is going to be compared to any pattern character during iteration s_1.

During iteration s_1, the algorithm performs some $r \geqq 1$ comparisons. By the definition of a light sequence, $r-1$ of these comparisons are matches and one is a mismatch. Thus, at the end of iteration s_1, the algorithm has the additional knowledge that $t[s_1+h_{\text{start}(s_1)}+1]=p[h_{\text{start}(s_1)}+1], \cdots, t[s_1+h_{\text{start}(s_1)+r-2}+1]=p[h_{\text{start}(s_1)+r-2}+1]$, $t[s_1+h_{\text{start}(s_1)+r-1}+1] \neq p[h_{\text{start}(s_1)+r-1}+1]$. Let $N=\{s_1+h_i+1: 1 \leqq i \leqq \text{start}(s_1)+r-1\}$. Notice that the algorithm performs $r=|N|-c(s_1)$ comparisons during iteration s_1. We bound this number by bounding $|N|$.

If s_2 is the dummy iteration, $|N| \leqq m \leqq s_2-s_1+c(s_2)$, since $s_2=n$, $s_1 \leqq n-m$ and $c(s_2)=0$. Therefore, the algorithm performs at most $s_2-s_1+c(s_2)-c(s_1)$ character comparisons during iteration s_1 paying for s_2-s_1 comparisons.

Assume s_2 is not the dummy iteration, i.e., $s_2 \leqq n-m$. We consider two mutually exclusive cases: $\text{start}(s_1)+r-1 \leqq nd$ and $\text{start}(s_1)+r-1 > nd$. For brevity, let $\hat{h}$ denote $h_{\text{start}(s_1)+r-1}$.

Assume that $\text{start}(s_1)+r-1 > nd$. By definition of a light sequence, this case can happen only if $h_{nd} < m-1$. Otherwise s_1 matches a suffix of the pattern. Moreover, we must have $\text{start}(s_1)+r-1=nd+1$, $h_{nd+1}=m-1$ and $t[s_1+m] \neq p[m]=p[h_{nd+1}+1]$. The algorithm shifts the pattern over the text by $r\text{min}(m-1)=m>m-1$ positions. Thus, $s_2=s_1+m$. Obviously, $|N| \leqq m=s_2-s_1$ and $c(s_2)=0$. Therefore, the number of character comparisons performed by the algorithm during iteration s_1 is at most $s_2-s_1+c(s_2)-c(s_1)$. Moreover, since none of the text positions in $[s_1+1, s_2]$ is busy (by invariant 1), they can pay for $c(s_1)+r-c(s_2)=|N|$ comparisons and become busy at the end of iteration s_1. Obviously, s_2 satisfies invariant 1 when it starts.

Let us consider now the case $\text{start}(s_1)+r-1 \leqq nd$. Since $k\text{min}(\hat{h})$ is defined and $t[s_1+\hat{h}+1] \neq p[\hat{h}+1]$, we have that $s_2=s_1+k\text{min}(\hat{h})$. We divide the set N into four disjoint subsets N_{nomatch}, N_{busy}, N_{holes}, N_{noholes} defined as follows.

- $N_{\text{nomatch}}=\{s_1+\hat{h}+1\}$,
- $N_{\text{busy}}=\{i \in N: i \leqq s_2\}$.
- N_{holes} is the set of i's in $N-\{s_1+\hat{h}+1\}$ such that $i>s_2$ and each i is aligned with a hole of the pattern at the beginning of iteration s_2.
- N_{nohole} is the set of i's in $N-\{s_1+\hat{h}+1\}$ such that $i>s_2$ and each i is aligned with a nohole of the pattern at the beginning of iteration s_2.

At the beginning of iteration s_2, Algorithm SM sets variable $e=\text{start}(s_2)$ to first $(\hat{h}-k\text{min}(\hat{h}))$. We claim that

$$|N_{\text{noholes}}|=\text{start}(s_2)-1=c(s_2). \tag{3}$$

Consider j, one of the first $c(s_2)$ noholes in the pattern not compared in iteration s_2. By Fact 4, $j+k\min(\hat{h})$ is a nohole in iteration s_1. Thus, in iteration s_1, j is aligned with an element of N_{noholes}. Conversely, each element of N_{noholes} is aligned with a nohole of the pattern in both iterations and the nohole in iteration s_2 is one of the first $c(s_2)$ of the pattern ($\text{start}(s_2)=\text{first}(\hat{h}-k\min(\hat{h}))$). Thus, (3) holds.

We claim that

$$|N_{\text{holes}}| \leqq s_2 - s_1 - 1 - |N_{\text{busy}}|. \tag{4}$$

Indeed, consider $p[1, \hat{h}]$ and let J denote the set of positions j in this string such that $k\min(\hat{h}) < \hat{h}, j$ is nohole and $j-k\min(\hat{h})$ is a hole. At the end of iteration s_1, each $j \in J$ has a match with exactly one text position in N_{holes} and vice versa. Thus, $|N_{\text{holes}}| = |J|$. If $k\min(\hat{h}) \leqq l$, $\hat{h} \leqq l$ by Fact 3 and $|J| = |N_{\text{busy}}| = 0$ since there cannot be any noholes in $p[1, l]$. Thus (4) obviously holds. Assume now that $k\min(\tilde{h}) > l$. From Lemma 5 (with $k = k\min(\hat{h})$), we know that each $j \in J$ is reachable from a distinct hole q in $p[2, k\min(\hat{h})]$. The number of holes in $p[1, k\min(\hat{h})]$ is $s_2 - s_1 - |N_{\text{busy}}| = k\min(\hat{h}) - |N_{\text{busy}}|$, since by definition, the elements of N_{busy} are exactly those aligned with noholes in $p[1, k\min(\hat{h})]$. Thus, $|N_{\text{holes}}| = |J| \leqq s_2 - s_1 - 1 - |N_{\text{busy}}|$ and (4) follows. We notice that the holes in $p[2, k\min(\hat{h})]$ were aligned with free positions at the beginning of iteration s_1, by invariant 1. Such text positions were ignored during iteration s_1 because s_1 tried to match noholes first and $\text{start}(s_1) + r - 1 \leqq nd$. Thus, there are enough free text positions from $s_1 + 2$ to s_2 to pay for the comparisons that involved text positions in N_{holes}. That is, at the end of iteration s_1 all text positions in N_{holes} are free, provided that the text positions aligned with holes in $p[2, k\min(\hat{h})]$ during iteration s_1 are declared busy.

We also observe that position $s_1 + 1$ is free at the beginning of iteration s_1 since it is aligned with $p[1]$ (a hole) and Invariant 1 holds. Therefore, it can pay for the mismatch. Thus, text position $s_1 + \hat{h} + 1$ is still free provided that $s_1 + 1$ becomes busy at the end of iteration s_1. All text positions in N_{busy} are declared busy at the end of iteration s_1.

We remark that all text positions declared busy at the end of iteration s_1 were either free at the beginning of this iteration or they were part of the $c(s_1)$ matches that s_1 inherited from previous iterations. Moreover, they are not going to be considered again by Algorithm SM after iteration s_1.

Using the identities (3), $|N_{\text{nomatch}}| = 1$, and inequality (4), it follows that $|N| \leqq s_2 - s_1 + c(s_2)$. Since $c(s_1)$ of those comparisons were performed during previous iterations, the algorithm performs at most $s_2 - s_1 + c(s_2) - c(s_1)$ comparisons during iteration s_1. Moreover, $c(s_2)$ of them are left unpaid by s_1.

Iteration s_2 satisfies Invariant 1 when it starts. Indeed, all text positions in $t[s_2+1, s_2+h_{\text{start}(s_2)}]$ aligned with holes of the pattern are free. Such text positions are a subset of N_{holes}. All text positions in $t[s_2+1, s_2+h_{\text{start}(s_2)}]$ aligned with noholes of the pattern match the corresponding characters. These positions are in N_{noholes}. All text positions in $t[s_2+h_{\text{start}(s_2)}+1, n]$ are free because either they were free at the beginning of s_1 and the algorithm did not consider them during s_1, or free text positions in $t[s_1+1, s_2]$ paid (by becoming busy) for their comparisons.

The analysis of the other iterations in the sequence is similar. Consequently, iteration s_i makes r_i comparisons and charges $r_i + c(s_i) - c(s_{i+1})$ $(s_{y+1} = u_1)$ to free positions in $[s_i + 1, u_1]$. Summing up these charges we get $\sum_{i=1}^{y} r_i + c(s_1) - c(u_1) \leqq u_1 - s_1$, and hence $\sum_{i=1}^{y} r_i \leqq u_1 - s_1 + c(s_1) - c(u_1)$. Moreover, Invariant 1 holds for u_1 inductively. Therefore we have the following Lemma.

LEMMA 8. *Let $s_1, s_2, \cdots, s_y$ be a light sequence and let u_1 be the iteration that starts the succeeding heavy sequence or the dummy iteration. Assuming that s_1 satisfies Invariant 1, Algorithm SM performs at most $u_1 - s_1 + c(u_1) - c(s_1)$ character comparisons during the execution of such a light sequence. Moreover, iteration u_1 satisfies Invariant 1.*

5.2. The analysis of heavy sequences. Consider a heavy sequence $u_1, u_2, \cdots, u_g$ and let s_1 be the iteration succeeding it. We estimate the total number of comparisons that the algorithm performs during the execution of such a sequence. We also show that s_1 satisfies Invariant 1 if it is not the dummy iteration.

We partition the sequence $u_1, u_2, \cdots, u_g$ into d consecutive subsequences $U_1, U_2, \cdots, U_d$ as follows. Let $u_{i_1} < u_{i_2} < \cdots < u_{i_{d-1}}$ be the iterations among $u_1, u_2, \cdots, u_g$ that match a suffix of the pattern. Then, $U_1 = \{u_1, \cdots, u_{i_1}\}$, $U_2 = \{u_{i_1+1}, \cdots, u_{i_2}\}$, $U_d = \{u_{i_{d-1}+1}, \cdots, u_g\}$. We denote the first and last element in each sequence U_j as first (U_j) and last (U_j), respectively. For each j, $1 \leqq j \leqq d$, last (U_j) can be thought of as being m positions to the left of a boundary line on the text that we denote by $\text{line}_j = \text{last}(U_j) + m$. Moreover, we set $\text{line}_0 = u_1$. Note that after iteration last (U_j), $1 \leqq j < d$, this boundary line is never crossed to the left because the algorithm sets $t$last to line_j, i.e., $t\text{last} = \text{line}_j$ when iteration last (U_j) ends. We bound the number of character comparisons that the algorithm can perform between two consecutive boundary lines during the given heavy sequence.

We assume that u_1 satisfies Invariant 1 when it starts. Since u_1 starts a heavy sequence, it matches a suffix of the pattern. Thus, $U_1 = \{u_1\}$. By Invariant 1, there are $c(u_1)$ noholes in the pattern that match the text characters in $t[\text{line}_0 + 1, \cdots, \text{line}_1]$ aligned with them at the beginning of iteration u_1. Such positions are not going to be considered by the algorithm during u_1, since $\text{start}(u_1) = c(u_1) + 1$. Moreover, the remaining text positions in $t[\text{line}_0 + 1, n]$ are free. During iteration u_1, in the worst case, the algorithm can compare all other symbols of $p[1, m]$. In any case, the pattern is shifted $u_2 - u_1 \geqq z$ positions to the right over the text. Fact 6 follows from these observations.

FACT 6. *Assume that u_1 satisfies Invariant 1. The number of character comparisons performed by the algorithm during u_1, the first iteration of a heavy sequence, is bounded by $m - c(u_1)$.*

We now derive an upper bound on the number of comparisons performed by the algorithm during iterations in U_j, $1 < j \leqq d$. For each $u \in U_j$, $1 < j \leqq d$, let $C(u)$ denote the set of text positions in $[\text{line}_{j-1} + 1, \text{line}_j]$ that are aligned with noholes in the pattern at the beginning of iteration u and that have been successfully matched against pattern characters during iterations in the heavy sequence preceding u. Of course, $C(u)$ may be the empty set. (Unlike $c(u)$, $|C(u)|$ only counts text positions between two lines that are aligned with noholes at the beginning of u.)

LEMMA 9. *Assume that $|U_j| > 1$ for some j, $1 < j \leqq d$. The number of character comparisons performed by the algorithm during an iteration $u \in U_j$, $u < \text{last}(U_j)$, is bounded by* $\max(1, \lfloor (u' - u)/2 \rfloor) + |C(u')| - |C(u)|$, *where u' is the iteration succeeding u in U_j.*

Proof. We give a proof for the case in which U_2 satisfies the hypothesis of the lemma. The proof can be immediately extended to the case $j > 2$.

Consider an iteration $u \in U_2$, $u < \text{last}(U_2)$, and let u' be the iteration succeeding it in U_2. Iteration u inherits $|C(u)|$ matches in $[\text{line}_1 + 1, \text{line}_2]$ from previous iterations in U_2. (Recall that all comparisons before U_2 were to the left of line_1.) Since u fails to match any suffix of the pattern, we must have $e \leqq nd$ at the end of iteration u. Thus, the algorithm performs q comparisons between noholes in $p[h_{\text{start}(u)} + 1, m]$ and the

corresponding text positions in $[\text{line}_1+1, n]$, skipping text positions in $C(u)$. One of those comparisons is a mismatch and $q-1$ are matches. Then, the pattern is shifted over the text by $u'-u=k\min(h_{\text{start}(u)+q-1})$ positions. Among the $|C(u)|+q-1$ text positions in $[\text{line}_1+1, \text{line}_2]$ that matched the corresponding pattern positions during iteration u, the algorithm records only $|C(u')|$ during iteration u' and such character comparisons will not be repeated during u'. We bound q by bounding $S=|C(u)|+q-|C(u')|-1$, i.e., the number of matches that the algorithm forgets in going from iteration u to u'. Let $k=u'-u$, $h=h_{\text{start}(u)+q-1}$, $b=\text{line}_1-u$. We notice that $b\leqq h$ (since after u_1 the algorithm only compares text symbols beyond line_1) and that $k=k\min(h)$ is a period of $p[1, h]$. We next show that $u'<\text{line}_1$ which implies $k<b$ for our choice of k and b. Indeed, by the way we partition the heavy sequence, $\text{last}(U_2)\geqq u'>u$ is the only iteration in U_2 that can increase the value of variable $t$last in algorithm SM when that iteration ends. Since $t\text{last}=\text{line}_1$ when iterations in U_2 start and $u'<t\text{last}(u')$ by definition of heavy sequence, we obtain $u'<\text{line}_1$.

Observe that the S matches that the algorithm forgets at the end of iteration u correspond to positions j in $p[1, h]$ such that j is a nohole, $j>b=\max(k, b)$, and $j-k$ is a hole. By Lemma 5, each of these pattern positions is reachable from a distinct hole in $p[1, k]$, $k=\min(k, b)$. Since our choice of k, b and h satisfies the hypotheses of Lemma 6, we obtain $S\leqq\max(0, \lfloor(k-2)/2\rfloor)$ from bound (2). Therefore $q=S+1+|C(u')|-|C(u)|\leqq\max(1, \lfloor(u'-u)/2\rfloor)+|C(u')|-|C(u)|$. □

We now consider the last iteration in U_j, $1<j<d$. Again, without loss of generality, we limit our discussion to the case $j=2$. Iteration $\text{last}(U_2)$ matches a suffix of the pattern since it advances $t$last to $\text{last}(U_2)+m$ and, by assumption, U_2 does not conclude a heavy sequence. Thus, in the worst case, the algorithm performs $\text{line}_2-\text{line}_1-|C(\text{last}(U_2))|$ comparisons of text positions in $[\text{line}_1+1, \text{line}_2]$, skipping text positions in $C(\text{last}(U_2))$. Then the pattern is shifted over the text by at least z positions and $t$last is set equal to line_2. Thus, all text positions in $[\text{line}_1+1, \text{line}_2]$ are not going to be considered again by the algorithm. Fact 7 summarizes these observations.

FACT 7. *The algorithm performs at most* $\text{line}_j-\text{line}_{j-1}-|C(\text{last}(U_j))|$ *character comparisons during iteration* $\text{last}(U_j)$, $1<j<d$.

If we had enough free text positions in $[\text{line}_{j-1}+1, \text{line}_j]$, $j<d$, to pay for all character comparisons performed during iterations in U_j, the analysis of heavy sequences would be the same as for light sequences. Unfortunately, $\text{last}(U_j)$ must use most of the free text positions in $[\text{line}_{j-1}+1, \text{line}_j]$ in order to pay for its own character comparisons. The remaining iterations in U_j pay for their character comparisons using a credit line. For any sequence $S=\{x_1; x_2; \cdots; x_{|S|}\}$ let $\text{credit}(S)=\sum_{k=2}^{|S|}\max(1, \lfloor(x_k-x_{k-1})/2\rfloor)$ (if $|S|\leqq 1$, let $\text{credit}(S)=0$).

LEMMA 10. *For each* U_j, $1<j<d$, *the total number of comparisons performed by the algorithm during iterations in* U_j *is bounded by*

$$\text{line}_j-\text{line}_{j-1}+\text{credit}(U_j). \tag{5}$$

Proof. By induction on j. For $j=2$, an upper bound on the total number of comparisons performed by the algorithm during iterations in U_2 is obtained by summing the bound in Lemma 9 for each $u\in U_2$, $u<\text{last}(U_2)$, and that in Fact 7. Since $|C(u_2)|=0$ (u_2 is the first iteration in U_2), this sum can be rewritten as (5). The proof is similar for $j>2$. □

We now consider the number of character comparisons performed by the algorithm during iteration u_g, the last iteration in the heavy sequence. Throughout our discussion, we assume that $u_g\neq u_1$, i.e., $g>1$ and $d>1$.

LEMMA 11. *Let $g>1$. The number of character comparisons that the algorithm performs during iteration u_g is bounded by*

$$s_1+c(s_1)-\text{line}_{d-1}-|C(u_g)| \tag{6}$$

when u_g is the last iteration of the algorithm, and by

$$s_1+c(s_1)-\text{line}_{d-1}-|C(u_g)|+\max\left(1,\left\lfloor\frac{\text{line}_{d-1}-u_g+1}{2}\right\rfloor\right) \tag{7}$$

when u_g is not the last iteration of the algorithm.

Proof. Iteration u_g inherits $|C(u_g)|$ matches in $[\text{line}_{d-1}+1, \textit{line}_d]$ from previous iterations in U_d. Obviously, if $U_d=\{u_g\}$ then $C(u_g)$ is empty. The algorithm performs q comparisons between text characters in $t[\text{line}_{d-1}+1, \text{line}_d]$ and the corresponding pattern characters. Then, the pattern is shifted over the text by at least $\text{line}_{d-1}-u_g$ positions (past $\text{line}_{d-1}=t\text{last}(s_1)$). We consider two cases according to whether or not u_g is the last iteration of the algorithm. Let $y=\text{start}(u_g)+q-1$, i.e., $p[h_y+1]$ is the last character of the pattern compared during this iteration.

Assume that u_g is the last iteration of the algorithm. Obviously, $q\leqq \text{line}_d-\text{line}_{d-1}-|C(u_g)|$. Recalling that $s_1=n$ when s_1 is the dummy iteration and that $\text{line}_d\leqq n$, we obtain that q is bounded by (6).

Assume u_g is not the last iteration of the algorithm. We consider two subcases: $y\geqq nd+1$ and $y\leqq nd$.

Subcase (i). $y\geqq nd+1$ and u_g not the last iteration of the algorithm: At the end of u_g, the pattern is shifted by $r\min(h_y)$, so $s_1-u_g=r\min(h_y)$. Since $y>nd$ and the algorithm processes the sequence h_i in increasing order of i until either $i=m+1$ or a mismatch is found, $p[h_{nd}+1]=t[u_g+h_{nd}+1]$. Recall that by definition of heavy sequence, $t\text{last}(s_1)\leqq s_1$. Thus, by Lemma 7, s_1 cannot have a suffix-prefix overlap with any iteration. It follows that $r\min(h_y)=m$ since $r\min(m-1)=m$ and $r\min(h_y)<m$ for $h_y<m-1$ would imply that s_1 has a suffix-prefix overlap with u_g. Thus, $s_1-u_g=m$. This equality implies that $s_1=\text{line}_d$ ($\text{line}_d=u_g+m$) and, in turn, that $c(s_1)=0$ (no comparisons made beyond line_d). Therefore, $q\leqq\text{line}_d-\text{line}_{d-1}-|C(u_g)|$ is bounded by $s_1+c(s_1)-\text{line}_{d-1}-|C(u_g)|$ and (7) holds.

Subcase (ii). $y\leqq nd$ and u_g is not the last iteration of the algorithm. Since $h_m=0$, $nd<m$ ($k\min(0)$ is never defined) and given the way the algorithm processes the sequence h_i, we have that the q comparisons made during iteration u_g result in $q-1$ matches and a mismatch. At the end of iteration u_g, the total number of matches between pattern positions and text positions in $[\text{line}_{d-1}+1, \text{line}_d]$ is $|C(u_g)|+q-1$. We bound q by bounding $|C(u_g)|+q$.

Let N be the set of text positions in $[\text{line}_{d-1}+1, \text{line}_d]$ that the algorithm has successfully matched with the corresponding pattern positions at the end of iteration u_g. $C(u_g)$ is included in N. Thus, $|N|+1=|C(u_g)|+q$. We partition N into three disjoint subsets as follows. N_{noholes} is the set of positions in N aligned with a nohole of the pattern at beginning of iteration s_1. N_{dead} is the set of positions in N falling in the interval $[\text{line}_{d-1}+1, s_1]$. N_{holes} is the set of positions in N aligned with holes at the beginning of iteration s_1. We claim that

$$N_{\text{holes}}\leqq s_1-\text{line}_{d-1}-|N_{\text{dead}}|+\max\left(0,\left\lfloor\frac{\text{line}_{d-1}-u_g-1}{2}\right\rfloor\right). \tag{8}$$

Indeed, let $h=h_y$, $k=k\min(h_y)\leqq h$ and $b=\text{line}_{d-1}-u_g$. Since $\text{line}_{d-1}=t\text{last}(s_1)\leqq s_1$ by definition of heavy sequence, $b=\text{line}_{d-1}-u_g\leqq s_1-u_g=k\min(h_y)=k$. Thus, $b\leqq k<h$ and k is a period of $p[1,h]$. Consider $p[1,h]$ and let J be the set of

positions j in that string such that $k\min(h) < j \leqq h$, j is a nohole and $j - k\min(h)$ is a hole. At the end of iteration u_g, each $j \in J$ has a match with exactly one text position in N_{holes} and vice versa. By Lemma 5, each position in J is reachable from a distinct hole in $p[1, k\min(h)]$. We can further partition J in two subsets J' and $\hat{J}$ such that $j' \in J'$ is reachable from a distinct hole in $p[1, b]$ and $\hat{j} \in \hat{J}$ is reachable from a distinct hole in $p[b+1, k\min(h)]$. We have $|\hat{J}| + |N_{\text{dead}}| \leqq k\min(h) - b = s_1 - \text{line}_{d-1}$, since $k\min(h) = s_1 - u_g$, $b = \text{line}_{d-1} - u_g$ and at the end of iteration u_g each position in N_{dead} has a match with a distinct nohole in $p[b+1, k\min(h)]$. Therefore, $|\hat{J}| \leqq s_1 - \text{line}_{d-1} - |N_{\text{dead}}|$. We use Lemma 6 with the same k, h and b to bound J'. Each $j' \in J'$ is such that $j' > k = \max(k, b)$, $j' - k$ is a hole and j' is reachable from a hole in $p[1, b]$, $b = \min(k, b)$. Therefore, our choice of J', k, b and h satisfies the hypotheses of Lemma 6 and since $b \leqq k$, by (1) $|J'| \leqq \max(0, \lfloor (b-1)/2 \rfloor) = \max(0, \lfloor (\text{line}_{d-1} - u_g - 1)/2 \rfloor)$. Using the bounds on $|\hat{J}|$ and $|J'|$ and the fact that $|N_{\text{holes}}| = |J| = |\hat{J}| + |J'|$, we obtain (8).

Recall that N_{noholes} is the set of positions in N aligned with noholes of the pattern at the beginning of iteration s_1. We have that $|N_{\text{noholes}}| = \text{start}(s_1) - 1 = c(s_1)$ (the proof is analogous to the one that (3) holds and is omitted).

Using the identities $|N_{\text{noholes}}| = \text{start}(s_1) - 1 = c(s_1)$, $|N| + 1 = |C(u_g)| + q$, and inequality (8) we obtain that q is bounded by (7). □

We can finally bound the total number of comparisons performed by the algorithm during iterations in U_d. Again, we must use credit in order to pay for some of the comparisons performed during iterations in U_d. For any sequence $S = \{x_1; x_2; \cdots; x_{|S|}\}$ let $\text{credit}^*(S) = \sum_{k=2}^{|S|-1} \max(1, \lfloor (x_k - x_{k-1})/2 \rfloor) + \max(1, \lfloor (x_{|S|} - x_{|S|-1} + 1)/2 \rfloor)$ (if $|S| \leqq 1$, $\text{credit}^*(S) = 0$).

LEMMA 12. *The number of character comparisons performed by the algorithm during iterations in U_d, $d > 1$, is bounded by*

$$s_1 + c(s_1) - \text{line}_{d-1} + \text{credit}^*(U_d \cup \{\text{line}_{d-1}\}), \tag{9}$$

when u_g is not the last iteration of the algorithm; by

$$s_1 + c(s_1) - \text{line}_{d-1} + \text{credit}(U_d) \tag{10}$$

when u_g is the last iteration of the algorithm.

Proof. Assume that u_g is not (is, respectively) the last iteration of the algorithm. A bound on the number of character comparisons performed during iterations in U_d is obtained by summing the bound in Lemma 9 for each $u \in U_d$, $u < u_g$, and bound (7) ((6), respectively) in Lemma 11. Since $|C(\text{first}(U_d))| = 0$, this sum can be rewritten as (9) ((10), respectively). □

We will repeatedly use the following lemma to bound the credit of each U_j, $1 < j \leqq d$. Its conditions are easily verified in each case.

LEMMA 13. *Consider the subsequence U_j, for some j, $1 < j \leqq d$. Let α and β be real numbers, $z \geqq \beta$. For each $u \in U_j \cup \{\text{line}_{j-1}\}$, $\lfloor \alpha(u - \text{first}(U_j) + \beta) \rfloor \leqq \lfloor \alpha((u - \text{last}(U_{j-1}))((z' + \beta)/m)) \rfloor$.*

Proof. Let left $= (u - \text{first}(U_{j-1}) + \beta)$ and let right $= (u - \text{last}(U_{j-1}))((z' + \beta)/m)$. We prove that left $\leqq$ right and therefore $\lfloor \alpha \text{left} \rfloor \leqq \lfloor \alpha \text{ right} \rfloor$ proving the lemma.

Since $\text{first}(U_j) - \text{last}(U_{j-1}) \geqq z$ by definition of $r\min$, left $\leqq u - \text{last}(U_{j-1}) - z + \beta$. Since $u \leqq \text{line}_{j-1}$ and $\text{line}_{j-1} - \text{last}(U_{j-1}) = m$, $u - \text{last}(U_{j-1}) \leqq m$. Moreover, $z \geqq \beta$ by assumption. Thus, left $\leqq (u - \text{last}(U_{j-1}) - (z - \beta)m/m) \leqq ((u - \text{last}(U_{j-1}))\ (1 - (z - \beta)/m)) = (u - \text{last}(U_{j-1}))((z' + \beta)/m) =$ right, since $m - z = z'$. □

We are now ready to prove the following lemma.

LEMMA 14. *Let $p[1, m]$ be a pattern with period z. Let $u_1, u_2, \cdots, u_g$ be a heavy sequence and let s_1 be the succeeding iteration or the dummy iteration. Assume that u_1*

satisfies Invariant 1. *If u_g is not the last iteration of the algorithm,* SM *performs at most*

$$s_1 - u_1 + c(s_1) - c(u_1) + \left\lfloor \left(\text{line}_{d-1} - u_1\right)\left(\frac{z'}{m}\right) \right\rfloor \tag{11}$$

character comparisons during the execution of such a heavy sequence. If u_g is the last iteration of the algorithm, the number of character comparisons is bounded by

$$s_1 - u_1 + c(s_1) - c(u_1) + \left\lfloor \left(u_g - u_1\right)\left(\frac{z'}{m}\right) \right\rfloor . \tag{12}$$

Moreover, s_1 satisfies Invariant 1.

Proof. We consider two cases: u_1 is the only iteration in the sequence and its logical complement. We first derive a bound on the total number of comparisons performed by the algorithm during the heavy sequence for both cases. Then, we show that s_1 satisfies Invariant 1.

Assume that u_1 is the only iteration in the heavy sequence. By Fact 6 the number of comparisons during iteration u_1 is at most $m - c(u_1)$. Moreover, $u_1 + m = \text{line}_1 = t\text{last}(s_1) \leqq s_1$ and $s_1 - u_1 \geqq m$ if $s_1 \leqq n - m$ and also if $s_1 > n - m$. Therefore, $m - c(u_1)$ is bounded by (11) when u_1 is not the last iteration of the algorithm and is bounded by (12) when u_1 is the last iteration of the algorithm.

Consider now the case in which u_1 is not the only iteration in the heavy sequence, i.e., $g > 1$ and $d > 1$. We recall that the number of character comparisons performed by the algorithm during U_1 is

$$m - c(u_1) = \text{line}_1 - u_1 - c(u_1). \tag{13}$$

For each U_j, $1 < j < d$, we now bound credit (U_j). Since $\max(1, \lfloor x/2 \rfloor) \leqq \lfloor x \rfloor$, for $x \geqq 1$, and the sequence U_j is strictly increasing, we have credit $(U_j) \leqq \lfloor \text{last}(U_j) - \text{first}(U_j) \rfloor$ and by Lemma 13 (with $u = \text{last}(U_j)$, $\alpha = 1, \beta = 0$) credit $(U_j) \leqq \lfloor \text{last}((U_j) - \text{last}(U_{j-1}))(z'/m) \rfloor$. Using (5), the number of character comparisons performed during U_j is bounded by

$$\text{line}_j - \text{line}_{j-1} + \left\lfloor (\text{last}(U_j) - \text{last}(U_{j-1}))\left(\frac{z'}{m}\right) \right\rfloor . \tag{14}$$

Using the fact that $\max(1, \lfloor (x+1)/2 \rfloor) \leqq \lfloor x \rfloor$, for $x \geqq 1$, (9), and Lemma 13 (with $u = \text{line}_{d-1}$, $\alpha = 1$, $\beta = 0$), the number of character comparisons performed during U_d, when u_g is not the last iteration of the algorithm, is bounded by

$$s_1 + c(s_1) - \text{line}_{d-1} + \left\lfloor (\text{line}_{d-1} - \text{last}(U_{d-1}))\left(\frac{z'}{m}\right) \right\rfloor . \tag{15}$$

Using (10) and the same arguments used to bound credit (U_j), the number of character comparisons performed during U_d, when u_g is the last iteration of the algorithm, is bounded by

$$s_1 + c(s_1) - \text{line}_{d-1} + \left\lfloor (u_g - \text{last}(U_{d-1}))\left(\frac{z'}{m}\right) \right\rfloor \tag{16}$$

When u_g is not (respectively, is) the last iteration of the algorithm and $g > 1$, a bound on the number of character comparisons performed by the algorithm during a heavy sequence is obtained by adding (13), (14) for each $1 < j < d$, and (15) (respectively, (16)). Such a sum is bounded by (11) (respectively, (12)) since the sequence last (U_j) is increasing.

We next show that s_1 satisfies Invariant 1 when it starts. Indeed, since u_1 satisfies Invariant 1, text positions in $[u_1 + 1, n]$ are either free or match all the noholes in

$p[1, h_{\mathrm{start}(u_1)}]$. Thus, text positions in $t[u_1+1, s_1]$ can become busy at the end of the heavy sequence. This amounts to s_1-u_1 text positions being declared busy. As for positions in $t[s_1+1, n]$, notice they are either free (they were free when u_1 started) or they match noholes in $p[1, h_{\mathrm{start}(s_1)}]$. Therefore s_1 satisfies Invariant 1 when it starts.

Notice that the positions declared busy at the end of s_1 can be used to pay for s_1-u_1 character comparisons. This "covers" $s_1-u_1-c(u_1)$ comparisons performed by the algorithm during the heavy sequence as well as the $c(u_1)$ character comparisons (all matches) that u_1 has inherited from the previous iteration. The $c(s_1)$ matches inherited from u_g by s_1 are left unpaid. The remaining comparisons are paid by a credit line. □

5.3. Putting the sequences together.

THEOREM 1. *Let $p[1, m]$ be a pattern with period z, $m=z+z'$, and $z'<z$, and let $t[1, n]$ be a text. Algorithm* SM *finds all occurrences of p in t in $O(n+m)$ time. In the worst case, the algorithm performs at most $n+\lfloor(n-m)(z'/(z+z'))\rfloor$ character comparisons.*

Proof. The $O(n+m)$ time bound is obvious. As for the number of character comparisons, we notice that iteration zero satisfies Invariant 1 trivially and, by Lemmas 8 and 14, each iteration starting either a light or a heavy sequence satisfies the same invariant and the bounds in Lemmas 8 and 14 hold. The bound of the theorem follows by summing up these bounds: The sum of the terms $u_1-s_1+c(u_1)-c(s_1)$ or $s_1-u_1+c(s_1)-c(u_1)$ is bounded by n and the sum of the terms $\lfloor(s_1-u_1)(z'/(z+z'))\rfloor$ or $\lfloor(u_g-u_1)(z'/(z+z'))\rfloor$ is bounded by $\lfloor(n-m)(z'/(z+z'))\rfloor$. Since $c(u)=0$ for u the dummy iteration, no comparison is left unpaid. □

The bound in Theorem 1 is tight for each value of z and $m=z+z'$, $z'<z$. Indeed, let $p[1, m]=a^{z'}b^{z-z'}a^{z'}$. Choose $n=cm$, c integer, and $t[1, n]=(p[1, m])^c$. Notice that the first nohole of the pattern is $z'+1$, since $k\min(z')=1$. We show that when the algorithm has discovered all occurrences of $p[1, m]$ in $t[1, jm]$, for some j, $1\leqq j\leqq c$, it has performed $jm+(j-1)z'$ character comparisons and the pattern is aligned with $t[(j-1)m+1, jm]$. Indeed, when the algorithm has discovered all occurrences of $p[1, m]$ in $t[1, m]$, it has performed m comparisons and the pattern is aligned with $t[1, m]$. Assume that the algorithm has just found all occurrences of $p[1, m]$ in $t[1, jm]$, for some j, $1\leqq j<c$, it has performed $jm+(j-1)z'$ character comparisons and the pattern is aligned with $t[(j-1)m+1, jm]$. In order to find the next occurrence, the algorithm shifts the pattern over the text by z positions and tests whether the first nohole of the pattern matches the corresponding text character, i.e., $p[z'+1]$ is compared with $t[jm+1]$. This comparison results in a mismatch ($p[z'+1]\neq p[1]=t[jm+1]$) and the pattern is shifted by $k\min(z')=1$. This is repeated $z'-1$ additional times, since $t[jm+2]=\cdots=t[jm+z']=p[1]\neq p[z'+1]$. Then, SM finally discovers, in m comparisons, that $p[1, m]=t[jm+1, (j+1)m]$. The number of comparisons during this phase is $m+z'$ which added to the $jm+(j-1)z'$ comparisons performed to find all occurrences of $p[1, m]$ in $t[1, jm]$ gives a total of $(j+1)m+jz'$ character comparisons. Moreover, the pattern is aligned with $t[jm+1, (j+1)m]$.

Thus, SM performs $cm+(c-1)z'=n+\lfloor(n-m)(z'/(z+z'))\rfloor$ character comparisons to find all occurrences of $p[1, m]$ in $t[1, n]$, $n=cm$.

6. Saving character comparisons. The analysis of Algorithm SM carried out in the previous section and the examples given there show that the main source of inefficiency (for character comparisons) is due to shifts by one during heavy sequences of iterations: Recall that we used $\max(1, \lfloor x/2\rfloor)\leqq x$, when $x\geqq 1$, where x is the distance between two consecutive iterations, or the length of the shift. This length can be 1 exactly when

we have a mismatch with $e=1$ and we shift by $k\min(h_1)=1$. Whenever $x \geqq 2$, we have $\max(1, \lfloor x/2 \rfloor) \leqq x/2$. In this section we describe a variation of Algorithm SM that takes care of this shortcoming by handling differently certain shifts by 1.

We give a somewhat redundant pseudo code version of the new algorithm SM′. It behaves like Algorithm SM, except that it handles differently the shifts by one during heavy sequences. Based on the length of the suffix-prefix overlap of the current iteration, it decides to behave as Algorithm SM or look for the regular expression $(p[1])^*p[l+1]$. Recall that l is the maximal integer such that $p[1]=\cdots=p[l]$ and $p[1]\neq p[l+1]$. We assume that $p \neq (p[1])^m$, $m>1$, because such a pattern can be easily searched for in n comparisons. We first give the pseudocode for the new algorithm and then discuss it in some detail.

```
        Algorithm SM'
    begin
        b ← 0; pstart ← 1; tlast ← 0; heavy = false
      repeat
        e ← pstart;
        if heavy = true and e = 1 and tlast > b+1 then
                    begin
                    —Find longest prefix of text[tlast+1, n] that matches—
                    —the regular expression p[1]*p[l+1]—
                    i ← tlast − b + 1;
                    while b+i ≦ n and text[b+i] = p[1] do i ← i+1;
                    if i ≦ l or text[b+i] ≠ p[l+1] then
                                              begin
                                                b ← b+i;
                                                pstart ← 1;
                                                tlast ← b;
                                                goto decide;
                                              end
                                          else
                                              begin
                                              pstart ← 2;
                                              tlast ← b+i;
                                              b ← tlast − (l+1);
                                              goto decide;
                                              end
                  end
        while e ≦ m and tlast < b + h_e + 1 and p[h_e + 1] = text[b + h_e + 1] do
        e ← e+1;
      —This part is as in Algorithm SM—
          Check whether or not an occurrence has been found
          set variables tlast, b, pstart accordingly
      —Decide how to set heavy—
      decide:     if b ≧ tlast then heavy ← false;
                                          else   heavy ← true;
        until b > n − m
      end
```

As before, u is an iteration if Algorithm SM′ sets variable $b=u$ during its execution. Notice that SM′ simulates SM for some iterations while it does not for others. By

simulation we mean that SM′ compares pattern positions against the corresponding text positions according to the sequence $h_1, \cdots, h_m$ defined in § 3. We point out that the sequence of iterations for algorithm SM′ is nondecreasing (for algorithm SM it is increasing) since b can be set to the same value twice. Indeed, assume that SM′ does not simulate SM during some iteration u (it executes the "new" part of code) and it tries to match a prefix of $t[t\text{last}+1, n]$ with $(p[1])^*p[l+1]$. If the match is successful, $t$last and b are updated and the numeric value of the next iteration u' may be equal to u. (If the match is not successful, then b is increased and $u < u'$.) However, notice that an iteration u in which SM′ does not simulate SM must be followed by an iteration u' in which it does, since either heavy = false or $p$start = 2 (the next value assigned to e) and the new part of code cannot be executed during u'. So, we can still distinguish between two consecutive iterations u and u' because either $|u-u'|>0$ or SM′ does not simulate SM during only one of them. Therefore, for each iteration u of SM′, we can still define start (u) and last (u) as in the previous section, provided that, when there is ambiguity, we give the additional information of whether SM′ simulates SM during u.

THEOREM 2. SM′ *finds all occurrences of the pattern in the text.*

Proof. For any given iteration, SM′ either simulates SM or searches for the regular expression $(p[1])^*p[l+1]$. We prove the theorem by showing that when each iteration u starts the following invariant holds.

INVARIANT 2. SM′ *has correctly found all occurrences of* $p[1, m]$ *in text positions* $[1, u]$ *and* $t[u+h_1+1]=p[h_1+1], \cdots, t[u+h_{p\text{start}-1}+1]=p[h_{p\text{start}-1}+1]$. *Moreover, if* $t\text{last}-u>0$, *then* $t[u+1, t\text{last}]=p[1, t\text{last}-u]$ *and* $t\text{last}-u \leqq h_{p\text{start}}$.

Initially $(u=0)$, Invariant 2 is trivially satisfied. Consider an iteration $u>0$; assume it satisfies the invariant. We show that u', the next iteration, also satisfies the invariant. We discuss only the case in which SM′ does not simulate SM during iteration u, since the other case is implied by the way SM works and its correctness.

Assume that SM′ does not simulate SM during iteration u. Thus, when u starts, $e=p\text{start}=1$ and $t\text{last}-u=t\text{last}-\text{b}>1$. Since h_1 is the first position of the pattern having $k$min defined, we have that $h_1=l$. By Invariant 2, $t[u+1, t\text{last}]=p[1, t\text{last}-u]$ and $t\text{last}-u\leqq h_1=l$. Thus, $t[u+1, t\text{last}]$ is a prefix of $p[1, l]$.

SM′ searches for a prefix of $t[t\text{last}+1, n]$ that matches the regular expression $(p[1])^*p[l+1]$. Assume that it scans $t[t\text{last}+1, k+1]$ during this search. If $k-u<l$ or $t[k+1]\neq p[l+1]$, then there can be no occurrence of the pattern in the text in $[u+1, k+1]$ and the algorithm correctly sets the next iteration $u'=k+1$, and this iteration must start from scratch ($p$start = 1 and $t$last = u'). Thus, u' satisfies the invariant. If $k-u\geqq l$ and $t[k+1]=p[l+1]$, then there can be no occurrence of the pattern in the text in $[u+1, k-l]$. However, there may be an occurrence in $k-l+1$ since $t[k-l+1, k+1]=p[1, l+1]$. The algorithm correctly sets $u'=k-l$, $t\text{last}=k+1$ and $p\text{start}=2$. Noting that $t\text{last}-u'\leqq l+1\leqq h_2$ and that $p[h_1+1]=p[l+1]=t[k+1]=t[u'+h_1+1]$, u' satisfies the invariant. □

The analysis of Algorithm SM′ proceeds along the same lines as the analysis of Algorithm SM. We divide the iterations into light and heavy sequences. The definition of both sequences is as in the previous section. We remark that Lemma 7 does not hold anymore since the condition $u<t\text{last}(u)$ does not necessarily imply that u has a suffix-prefix overlap with an iteration u' preceding it.

Again, for each iteration starting a sequence we maintain Invariant 1. Thus, we can still use Lemma 8 for the analysis of a light sequences. In order to bound the number of character comparisons performed by Algorithm SM′, we need to analyze heavy sequences again and to show that the iteration succeeding a heavy sequence satisfies Invariant 1.

6.1. The analysis of heavy sequences for Algorithm SM′. Consider a heavy sequence $u_1, u_2, \cdots, u_g$ and let s_1 be the iteration succeeding it. We partition the sequence $u_1, u_2, \cdots, u_g$ into d consecutive subsequences $U_1, U_2, \cdots, U_d$ as in the preceding section. Again, for each U_j, $1 \leqq j \leqq d$, we define a boundary line, $\text{line}_j = \text{last}(U_j) + m$. Moreover, we set $\text{line}_0 = u_1$. As before, last (U_j) is the only iteration in U_j that matches a suffix of the pattern, $1 \leqq j < d$.

We need the following observations about iterations in U_j, $1 < j \leqq d$, handling a shift by one.

LEMMA 15. *Each U_j, $1 < j \leqq d$ has at most one iteration u such that $e = 1$ when it starts. It is either the next to last or the last iteration in U_j. When $u = \text{line}_{j-1} - 1$, it is the last iteration in U_j. When $u < \text{line}_{j-1} - 1$, it cannot match any suffix of the pattern.*

Proof. Assume that there is more than one iteration in U_j such that $e = 1$ holds when it starts. Let u and u', $u \leqq u'$ be the first two. We consider two cases: $u = \text{line}_{j-1} - 1$ and $u < \text{line}_{j-1} - 1$ (we cannot have $u \geqq \text{line}_{j-1}$ since $u \in U_j$). Notice that when u starts, $t$last must be equal to line_{j-1} since no iteration in U_j preceding u has matched a suffix of the pattern or it had $e = 1$, an essential condition to modify the value of $t$last in the new code. Moreover, SM′ simulates SM at least up to the iteration preceding u.

Case $u = \text{line}_{j-1} - 1$. Since $t\text{last} = \text{line}_{j-1} = u + 1$ Algorithm SM′ simulates SM during the execution of iterations in U_j at least up, and including, u. Now, if u finds a mismatch before matching any suffix of the pattern, it shifts by at least up to 1 and does not update $t$last: the next iteration $\hat{u}$ is such that $\text{line}_{j-1} = t\text{last} = t\text{last}(\hat{u}) \leqq \hat{u}$. Therefore u concludes the heavy sequence. If u matches a suffix of the pattern, u concludes U_j. In any case, u' cannot exist and u is the last iteration in U_j.

Case $u < \text{line}_{j-1} - 1$. Since $t\text{last} = \text{line}_{j-1} > u + 1$, heavy = true and $e = 1$, Algorithm SM′ looks for the longest prefix of $t[\text{line}_{j-1} + 1, n]$ matching the regular expression $(p[1])^* p[l+1]$. If no prefix matches, u ends the heavy sequence and no u'exists. If a prefix matches, then $\hat{u} = t\text{last} - (l+1) = t\text{last}(\hat{u}) - (l+1)$, the iteration succeeding u, must shift by at least $l+1$. Thus, if u' exists we have $\hat{u} < u'$. Moreover, SM′ must simulate SM during $\hat{u}$. Now, if $\hat{u}$ does not match a suffix of the pattern, it concludes the heavy sequence. Indeed, $t\text{last} = t\text{last}(\hat{u})$ cannot be updated and the next iteration is $s \geqq \hat{u} + l + 1 = t\text{last}(\hat{u}) = t\text{last}(s)$. Therefore no u' exists. If $\hat{u}$ matches a suffix of the pattern, $\hat{u}$ is the last iteration in U_j and no u' exists. Therefore, u is either the next to last or the last iteration in U_j.

Iteration u can match a suffix of the pattern only if $p[1, m] = p[1]^l p[l+1]$ (see Algorithm SM′). But, recalling the definition of heavy sequence, we can have only heavy sequences consisting of one element for $p[1]^l p[l+1]$. Indeed, each iteration that matches a suffix of the pattern must then shift the pattern by $l+1$ positions. Thus, there is no suffix-prefix overlap between consecutive iterations implying that $b \geqq t$last always holds during the execution of SM′ for $p[1]^l p[l+1]$. So, variable heavy is always false, a contradiction since we are assuming that heavy = true when u starts. □

For each U_j, $1 < j \leqq d$, let so (U_j) denote the only iteration in U_j, if any, such that $e = 1$ and so $(U_j) < \text{line}_{j-1} - 1$ when it starts. Notice that so (U_j) is the only iteration in U_j in which SM′ does not simulate SM. We also need the following fact.

FACT 8. *Assume that* so (U_j) *exists in U_j, $1 < j \leqq d$, then the pattern must have $p[1, q] = p[m - q + 1, m] = p[1]^q$, for some q, $2 \leqq q \leqq l$.*

Proof. By the way we partition the heavy sequence, last (U_{j-1}) must match a suffix of the pattern. Moreover, so (U_j) is the first iteration in U_j that can advance $t$last. Therefore, so (U_j) has a suffix-prefix overlap with last (U_{j-1}). The length of the overlap is $\text{line}_{j-1} - \text{so}(U_j) \geqq 2$, since $\text{so}(U_j) < \text{line}_{j-1} - 1$ by hypothesis. Therefore, letting $q = \text{line}_{j-1} - \text{so}(U_j)$, $p[1, q] = p[m - q + 1, m]$, $q \geqq 2$. But $q \leqq l$ because the overlap with last (U_{j-1}) is at most l; otherwise the shift and e would be larger than 1. □

Assume that u_1 satisfies Invariant 1 when it starts. Since u_1 is the first iteration of the heavy sequence, Algorithm SM′ starts with heavy = false. This implies that SM′ simulates SM during the execution of u_1. Thus, Fact 6 holds.

We now estimate the number of comparisons that SM′ performs for iterations in U_j, $1<j\leqq d$.

LEMMA 16. *Consider an iteration* $u\in U_j$, *for some* j, $1<j\leqq d$. *Assume that* $u<\text{last}(U_j)$ *and that* $u\neq\text{so}(U_j)$, *if* $\text{so}(U_j)$ *exists. The number of character comparisons performed by* SM′ *during iteration u is bounded by* $\lfloor(u'-u)/2\rfloor+|C(u')|-|C(u)|$, *where* u' *is the iteration succeeding u in* U_j.

Proof. By the hypothesis of Lemma 16 and Lemma 15, no iteration in U_j up to, and including, u can start with a value of $e=1$. This implies that SM′ simulates SM on these iterations. Moreover $u'-u\geqq 2$, since none of those iterations can shift by 1. Therefore, the bound in Lemma 9 holds and it reduces to $\lfloor(u'-u)/2\rfloor+|C(u')|-|C(u)|$. □

We now consider $\text{last}(U_j)$ and $\text{so}(U_j)$, if it exists, for a given subsequence U_j, $1<j<d$. Then, we consider $\text{last}(U_d)$ and $\text{so}(U_d)$, if it exists.

FACT 9. *Consider a subsequence* U_j, *for some* j, $1<j<d$. *Assume that* $\text{so}(U_j)$ *does not exist. The number of character comparisons performed by* SM′ *during iteration* $\text{last}(U_j)$ *is bounded by* $\text{line}_j-\text{line}_{j-1}-|C(\text{last}(U_j))|$.

Proof. Since $\text{so}(U_j)$ does not exist, SM′ simulates SM during all iterations in U_j. Thus, the bound follows as in Fact 7. □

FACT 10. *Consider a subsequence* U_j, *for some* j, $1<j<d$, *and assume that* $\text{so}(U_j)$ *exists. Then* $\text{so}(U_j)$ *is not the last iteration of* U_j *and the number of character comparisons performed by* SM′ *during iterations* $\text{so}(U_j)$ *and* $\text{last}(U_j)$ *is bounded by* $\text{line}_j-\text{line}_{j-1}-|C(\text{so}(U_j))|+1$.

Proof. We note that $C(\text{so}(U_j))=\phi$ when $\text{so}(U_j)$ starts, because it starts with $e=1$. Since $\text{so}(U_j)<\text{line}_{j-1}-1$, Algorithm SM′ tries to find the longest prefix of $t[\text{line}_{j-1}+1,n]$ that matches the regular expression $(p[1])^*p[l+1]$. Since we are assuming that U_j does not conclude a heavy sequence, the algorithm must succeed in its task (otherwise it would set $t\text{last}=b$ and then heavy = false). Therefore, it sets $b=\text{last}(U_j)$ and $t\text{last}=\text{last}(U_j)+l+1$ and the text characters to the left of $\text{last}(U_j)+l+2$ will not be considered again by future iterations. By inspection of the pseudocode performing the search for $(p[1]^*)p[l+1]$, the algorithm performs $\text{last}(U_j)+l+2-\text{line}_{j-1}$ character comparisons, one of which is a mismatch in iteration $\text{so}(U_j)$. Now, iteration $\text{last}(U_j)$ performs at most $m-l-1$ character comparisons. Therefore, the total for the two iterations is $m+\text{last}(U_j)-\text{line}_{j-1}+1=\text{line}_j-\text{line}_{j-1}-|C(\text{so}(U_j))|+1$. □

We can now prove the following lemma:

LEMMA 17. *The number of character comparisons performed by Algorithm* SM′ *during* U_j, $1<j<d$, *is bounded by*

$$\text{line}_j-\text{line}_{j-1}+\left\lfloor(\text{last}(U_j)-\text{last}(U_{j-1}))\left(\frac{z'}{2m}\right)\right\rfloor, \tag{17}$$

if $\text{so}(U_j)$ *does not exist. Otherwise, it is bounded by*

$$\text{line}_j-\text{line}_{j-1}+\left\lfloor(\text{so}(U_j)-\text{last}(U_{j-1}))\left(\frac{z'+2}{2m}\right)\right\rfloor \tag{18}$$

when $|U_j|>2$; *and by*

$$\text{line}_j-\text{line}_{j-1}+1 \tag{19}$$

when $|U_j|=2$.

Proof. Assume that so (U_j) does not exist. A bound on the number of character comparisons performed during iterations in U_j is obtained by adding the bounds in Lemma 16, for each iteration $u \neq \text{last}(U_j)$, and the bound in Fact 9. This sum is bounded by $\text{line}_j - \text{line}_{j-1} + \lfloor(\text{last}(U_j) - \text{first}(U_j))/2\rfloor$, since $C(\text{first}(U_j)) = \phi$ and, in turn, by (17) since $\lfloor(\text{last}(U_j) - \text{first}(U_j))/2\rfloor \leqq \lfloor(\text{last}(U_j) - \text{last}(U_{j-1}))(z'/2m)\rfloor$ by Lemma 13 (with $u = \text{last}(U_j)$, $\alpha = \frac{1}{2}$ and $\beta = 0$).

Assume now that so (U_j) exists. If $|U_j| = 2$, bound (19) follows from Fact 10 and the fact that $C(\text{so}(U_j)) = \phi$. Consider now the subcase $|U_j| > 2$. A bound on the number of character comparisons performed by iterations in U_j is obtained by adding the bounds in Lemma 16, for each iteration $u < \text{so}(U_j)$, and the bound in Fact 10. This sum is bounded by $\text{line}_j - \text{line}_{j-1} + \lfloor(\text{so}(U_j) - \text{first}(U_j) + 2)/2\rfloor$, since $C(\text{first}(U_j)) = \phi$ and, in turn, by (18) since $z \geqq 2$ and $\lfloor(\text{so}(U_j) - \text{first}(U_j) + 2)/2\rfloor \leqq \lfloor(\text{so}(U_j) - \text{last}(U_{j-1}))((z'+2)/2m)\rfloor$ by Lemma 13 (with $u = \text{so}(U_j)$, $\alpha = \frac{1}{2}$ and $\beta = 2$). □

We now consider so (U_d) and last (U_d), when so (U_d) exists. We do not discuss the case in which so (U_d) does not exist, since SM′ simulates SM for all iterations in U_d and a bound on the number of character comparisons performed by SM′ can be obtained using Lemma 12.

When so (U_d) starts, algorithm SM′ tries to find the longest prefix of $t[\text{line}_{d-1} + 1, n]$ of length at least $l + 1 - \text{line}_{d-1} + \text{so}(U_d)$ that matches the regular expression $(p[1])^* p[l+1]$ (SM′ does not simulate SM during so (U_j)). Assume that the algorithm "scans" $t[\text{line}_{d-1}, k]$ during this search. It can be easily seen that the number of character comparisons is at most $k - \text{line}_{d-1} + 1$. If so (U_d) is the last iteration of the algorithm or in U_d (end of text or search not successful), $k = s_1$ and $c(s_1) = 0$, where s_1 is the next iteration (recall our conventions about the dummy iteration). Otherwise (search successful and not end of text), $k = \text{last}(U_d) + l + 1$. In all cases, text characters to the left of $k + 1$ will not be considered again, since SM′ sets $t\text{last} = k$. So, we have Fact 11.

FACT 11. *Assume that* so (U_d) *exists. The number of character comparisons performed by Algorithm* SM′ *during* so (U_d) *is bounded by*

$$s_1 + c(s_1) - \text{line}_{d-1} + 1 \tag{20}$$

when so (U_d) *concludes either the sequence or the algorithm, and by*

$$\text{last}(U_d) + l - \text{line}_{d-1} + 2 \tag{21}$$

in all other cases.

Consider last (U_d), when so (U_d) exists. Algorithm SM′ behaves like Algorithm SM when matching pattern positions with the corresponding text positions in $[\text{last}(U_d) + l + 2, \text{last}(U_d) + m]$. We consider two cases: last (U_d) is not the last iteration of the algorithm and its logical complement. In any case, let q be the number of character comparisons that SM′ performs between $t[\text{last}(U_d) + l + 2, \text{line}_d]$ and the corresponding part of the pattern during last (U_d).

Assume that last (U_d) does not conclude the algorithm. Since it concludes the heavy sequence, the q comparisons result in $q - 1$ matches and a mismatch (last (U_j) can match the pattern and conclude the heavy sequence only when $z' = 0$, but then the heavy sequence consists of one element only and so (U_j) could not exist). Then, the pattern is shifted over the text by at least $l + 1 \geqq 2$ positions (recall that last (U_d) shifts by at least $l + 1$). We are interested in finding an upper bound on q. We consider two subcases: $s_1 - \text{last}(U_d) = m$ and $s_1 - \text{last}(U_d) < m$.

If $s_1 - \text{last}(U_d) = m$, we have $s_1 = \text{line}_d = \text{last}(U_d) + m$. Since no text character in $t[\text{line}_d, n]$ has been considered by the algorithm during iterations preceding s_1, we have that $c(s_1) = 0$, and q is obviously bounded by $s_1 + c(s_1) - (\text{last}(U_d) + l + 1)$.

Assume now that $s_1 - \text{last}(U_d) < m$. Let N be the set of text positions in $[\text{last}(U_d) + l+2, \text{line}_d]$ that the algorithm has successfully matched with the corresponding pattern positions at the end of iteration $\text{last}(U_d)$. Thus, $|N|+1=q$. We partition N in three disjoint subsets as follows. N_{noholes} is the set of positions in N aligned with a nohole of the pattern at the beginning of iteration s_1. N_{dead} is the set of positions in N falling in the interval $[\text{last}(U_d)+l+2, s_1]$. N_{holes} is the set of positions in N aligned with holes at the beginning of iteration s_1. We bound q by bounding $|N_{\text{holes}}|+|N_{\text{dead}}|$ and $|N_{\text{noholes}}|$.

Let $h = h_{\text{start}(\text{last}(U_d))+q-1}$. Consider $p[1, h]$ and let J be the set of positions j in that string such that $k\min(h) < j \leqq h$, j is a nohole, and $j - k\min(h)$ is a hole. Since $s_1 - \text{last}(U_d) = k\min(h) \geqq l+1$ ($\text{last}(U_d)$ shifts by at least $l+1$) and $k\min(h) < j$, for each $j \in J$, each $j \in J$ has a match with exactly one text position in N_{holes} and vice versa at the end of iteration $\text{last}(U_d)$. Therefore $|N_{\text{holes}}| = |J|$. By Lemma 5 (with $k = k\min(h)$), each nohole in J is reachable from a distinct hole in $p[l+2, k\min(h)]$. Each of the noholes in $p[l+2, k\min(h)]$ has a match with exactly one text position in $|N_{\text{dead}}|$ at the end of iteration $\text{last}(U_d)$ (recall the definition of N_{dead}). Thus, $|N_{\text{holes}}|+|N_{\text{dead}}| \leqq k\min(h)-l-1 = s_1 - \text{last}(U_d)-l-1$. Obviously, $|N_{\text{noholes}}| = \text{start}(s_1)-1 = c(s_1)$. Therefore, $q = |N|+1 = |N_{\text{holes}}|+|N_{\text{dead}}|+|N_{\text{noholes}}|+1 \leqq s_1 + c(s_1) - (\text{last}(U_d)+l+1)+1$.

If $\text{last}(U_d)$ is the last iteration of the algorithm, q is obviously bounded by $n-(\text{last}(U_d)+l+1) \leqq s_1 + c(s_1) - (\text{last}(U_d)+l+1)$, since $t\text{last} = \text{last}(U_d)+l+1$ when U_d starts and no character in $[1, t\,\text{last}]$ is ever considered again.

These observations can be summarized as follows.

FACT 12. *Assume that* so (U_d) *exists. The number of character comparisons performed by the algorithm during iteration* last (U_d) *is bounded by*

$$s_1 + c(s_1) - (\text{last}(U_d)+l+1) \tag{22}$$

when last (U_d) *concludes the algorithm or* $s_1 - \text{last}(U_d) = m$, *and by*

$$s_1 + c(s_1) - (\text{last}(U_d)+l+1)+1 \tag{23}$$

in all other cases.

LEMMA 18. *The number of character comparisons performed by Algorithm* SM′ *during iterations in* U_d *is bounded by*

$$s_1 + c(s_1) - \text{line}_{d-1} + \left\lfloor (\text{line}_{d-1} - \text{last}(U_{d-1}))\left(\frac{z'+1}{2m}\right)\right\rfloor \tag{24}$$

when u_g *does not conclude the algorithm and when* so (U_d) *does not exist. It is bounded by*

$$s_1 + c(s_1) - \text{line}_{d-1} + \left\lfloor (u_g - \text{last}(U_{d-1}))\left(\frac{z'}{2m}\right)\right\rfloor \tag{25}$$

when u_g *concludes the algorithm and when* so (U_d) *does not exist. It is bounded by*

$$s_1 + c(s_1) - \text{line}_{d-1} + \left\lfloor (\text{line}_{d-1} - \text{last}(U_{d-1}))\left(\frac{z'+2}{2m}\right)\right\rfloor \tag{26}$$

in all other cases.

Proof. We first consider the case in which u_g does not conclude the algorithm and so (U_d) does not exist. Algorithm SM′ simulates SM on iterations in U_d and a bound on the number of character comparisons performed by SM′ during iterations

in U_d is given by (9) in Lemma 12. We will obtain (24) from this bound by bounding credit*. Notice that the sequence $U_d \cup \{\text{line}_{d-1}\}$ is strictly increasing.

Notice that $\text{line}_{d-1} - \text{last}(U_d) + 1 \geqq 2$ ($\text{line}_{d-1} - \text{last}(U_d) \geqq 1$ by definition of heavy sequence and the case being considered). Moreover, for each u and u' in U_d, $\text{first}(U_d) \leqq u < u' \leqq \text{last}(U_d)$, $u' - u \geqq 2$ since no iteration in U_d, except possibly the last one, has $\text{start}(u) = 1$ (shift by 1 in case of mismatch). Therefore, $\text{credit}^*(U_d \cup \{\text{line}_{d-1}\}) \leqq \lfloor(\text{line}_{d-1} - \text{first}(U_d) + 1)/2\rfloor$. By Lemma 13 (with $u = \text{line}_{d-1}, \alpha = \frac{1}{2}, \beta = 1$), $\text{credit}^*(U_d \cup \{\text{line}_{d-1}\})$ is further bounded by $\lfloor(\text{line}_{d-1} - \text{last}(U_{d-1}))((z'+1)/2m)\rfloor$. Using (9), we obtain (24).

Consider the case in which u_g concludes the algorithm and so (U_d) does not exist. Algorithm SM′ simulates SM on iterations in U_d and a bound on the number of character comparisons performed by SM′ during iterations in U_d is given by (10) in Lemma 12. Employing the arguments used to bound credit*, yields $\text{credit}(U_d) \leqq \lfloor(u_g - \text{last}(U_{d-1}))(z'/2m)\rfloor$. Using (10), we obtain (25).

Consider the remaining cases. That is, $\text{so}(U_d)$ exists. Recall that $\text{so}(U_d) < \text{line}_{d-1} - 1$. Let $\hat{U} = \{x_1, \cdots, x_s\}$ be U_d up to $\text{so}(U_d)$, i.e., $x_s < \text{so}(U_d)$. Notice that $x_i - x_{i-1} \geqq 2$, $i > 1$, since no iteration in $\hat{U}$ has $\text{start}(x_i) = 1$. A bound on the number of character comparisons performed by SM′ during iterations in $\hat{U}$ is obtained by adding the bound in Lemma 16 for each $x_i \in \hat{U}$, $1 < i \leqq s$. This sum is bounded by

$$\left\lfloor \frac{\text{so}(U_d) - \text{first}(U_d)}{2} \right\rfloor. \tag{27}$$

If $\text{so}(U_d)$ is the last iteration of the algorithm or of U_d, the sum of (20) and (27) gives $s_1 + c(s_1) - \text{line}_{d-1} + 1 + \lfloor(\text{so}(U_d) - \text{first}(U_d))/2\rfloor \leqq s_1 + c(s_1) - \text{line}_{d-1} + \lfloor(\text{line}_{d-1} - \text{first}((U_d)))/2\rfloor$, since $\text{so}(U_d) \leqq \text{line}_{d-1} - 2$ by assumption. We derive the bound in (26) since $\lfloor(\text{line}_{d-1} - \text{first}(U_d))/2\rfloor \leqq \lfloor(\text{line}_{d-1} - \text{last}(U_{d-1}))(z'/2m)\rfloor$ by Lemma 13, with $u = \text{line}_{d-1}$, $\alpha = \frac{1}{2}$ and $\beta = 0$.

If $\text{so}(U_d)$ is neither the last iteration of the algorithm nor of U_d, the sum of (21), (23), and (27) gives $s_1 + c(s_1) - \text{line}_{d-1} + 2 + \lfloor(\text{so}(U_d) - \text{first}(U_d))/2\rfloor \leqq s_1 + c(s_1) - \text{line}_{d-1} + \lfloor(\text{line}_{d-1} - \text{first}((U_d)) + 2)/2\rfloor$, since $\text{so}(U_d) \leqq \text{line}_{d-1} - 2$ by assumption. We derive the bound in (26) since $\lfloor(\text{line}_{d-1} - \text{first}(U_{d-1}) + 2)/2\rfloor \leqq \lfloor(\text{line}_{d-1} - \text{last}(U_{d-1}))((z'+2)/2m]\rfloor$ by Lemma 13, with $u = \text{line}_{d-1}$, $\alpha = \frac{1}{2}$ and $\beta = 2$, and $z \geqq 2$. □

LEMMA 19. *Consider a heavy sequence $u_1, u_2, \cdots, u_g$ and let s_1 be the iteration succeeding it or the dummy iteration. The number of character comparisons performed by algorithm* SM′ *during the heavy sequence is bounded by*

$$s_1 + c(s_1) - u_1 - c(u_1) + \left\lfloor \frac{\text{line}_{d-1} - u_1}{l+1} \right\rfloor \tag{28}$$

when $p[1, m] = p[1]^l p[l+1]^l p[1]^l$, $l \geqq 2$, and $p[1] \neq p[l+1]$. It is bounded by

$$s_1 + c(s_1) - u_1 - c(u_1) + \left\lfloor (\text{line}_{d-1} - u_1)\left(\frac{z'+1}{2m}\right) \right\rfloor \tag{29}$$

when the pattern does not satisfy $p[1] = p[2] = p[m-1] = p[m]$ and u_g does not conclude the algorithm. It is bounded by

$$s_1 + c(s_1) - u_1 - c(u_1) + \left\lfloor (u_g - u_1)\left(\frac{z'}{2m}\right) \right\rfloor \tag{30}$$

when the pattern does not satisfy $p[1]=p[2]=p[m-1]=p[m]$ *and* u_g *concludes the algorithm. It is bounded by*

$$s_1+c(s_1)-u_1-c(u_1)+\left\lfloor (\text{line}_{d-1}-u_1)\left(\frac{z'+2}{2m}\right)\right\rfloor \tag{31}$$

in all the other cases. Moreover, s_1 *satisfies Invariant* 1 *when it starts.*

Proof. We now consider four cases, each corresponding to one of the bounds.

Case $p[1, m]=p[1]^l p[l+1]p[1]^l$, $l\geqq 2$, and $p[1]\neq p[l+1]$. The contribution of each U_j, $1<j<d$ to the total number of character comparisons depends on how many iterations it contains. We claim that $|U_j|\leqq 2$, $1<j<d$. Indeed, last (U_{j-1}) matches a suffix of the pattern and shifts the pattern by at least $z=l+1$ setting $b=$ first (U_j) and $t\text{last}=\text{line}_{j-1}=\text{last}(U_{j-1})+m$. Thus, $\text{line}_{j-1}-\text{first}(U_j)\leqq l$. Since the first nohole in the pattern is $l+1$, it is aligned with a text position past line_{j-1} when first (U_j) starts. Therefore, start (first $(U_j))=e=1$. If $t\text{last}-b=\text{line}_{j-1}-\text{first}(U_j)=1$, then SM′ simulates SM during first (U_j) and this iteration must match a suffix of the pattern; otherwise it would conclude the heavy sequence. Thus $|U_j|=1$ when $t\text{last}-b=\text{line}_{j-1}-\text{first}(U_j)=1$ and the number of character comparisons is bounded by $\text{line}_j-\text{line}_{j-1}$. If $t\text{last}-b=\text{line}_{j-1}-\text{first}(U_j)>1$, then SM′ does not simulate SM during first (U_j) as this iteration looks for the longest prefix of $t[\text{line}_{j-1}+1, n]$ that matches the regular expression $(p[1])^*p[l+1]$. It must find it, otherwise it would conclude the sequence. Now, second (U_j) must match a suffix of the pattern. Thus $|U_j|=2$ when $t\text{last}-b=\text{line}_{j-1}-\text{first}(U_j)>1$ and by Lemma 17, the number of character comparisons for $|U_j|=2$ is bounded by (19), i.e., $\text{line}_j-\text{line}_{j-1}+1$.

Since $|U_1|=1$, the sum of character comparisons performed during iterations in $U_1,\cdots,U_{d-1}$ is bounded by $\text{line}_{d-1}-u_1+(d-2)$.

Consider U_d. Notice that $|U_d|\leqq 2$ and that first (U_d) must have start (first $(U_d))=e=1$. The proof is the same as the one showing $|U_j|\leqq 2$ and that first (U_j) must have start (first $(U_j))=e=1$, $1<j<d$. We claim that the number of character comparisons for iterations in U_d is bounded by $s_1+c(s_1)-\text{line}_{d-1}+1$. This follows from Fact 11, bound (20), when $|U_d|=1$ and so (U) exists.

When $|U_d|=1$ and so (U_d) does not exist, SM′ simulates SM during first (U_d). This implies that first $(U_d)=b=t\text{last}-1=\text{line}_{d-1}-1$. Notice that, for the specific pattern we are considering, if first (U_d) matches a suffix of the pattern the next iteration must have a suffix-prefix overlap with first (U_d) and the heavy sequence would continue. Thus, first (U_d) cannot match any suffix of the pattern. For the specific pattern we are considering, $h_1=l$ and $h_2=m-1$ in the sequence $h_1,\cdots,h_m$. So, the mismatch must either be with $p[l+1]$ or with $p[2l+1]$. In either case, the shift is at least as large as the number of character comparisons performed during first (U_d) and $c(s_1)=0$. So, we obtain a bound of $s_1-\text{first}(U_d)=s_1-\text{line}_{d-1}+1=s_1+c(s_1)-\text{line}_{d-1}+1$. When $|U_d|=2$, so (U_d) must exist, otherwise Lemma 15 would be contradicted. The claimed bound is obtained as the sum of (21) and (22), since $s_1-\text{last}(U_d)=m$ (otherwise last (U_d) would completely match a suffix of the pattern).

Therefore, the number of character comparisons performed during this heavy sequence is bounded by $s_1+c(s_1)-u_1-c(u_1)+(d-1)$, where $d-1$ is the number of iterations in the heavy sequence that completely matched a suffix of the pattern (for $c(u_1)=0$). Since after such match the pattern is shifted by at least $l+1$ positions, $d-1\leqq\lfloor(\text{line}_{d-1}-u_1)/l+1\rfloor$ and we obtain (28).

Case. The pattern does not satisfy $p[1]=p[2]=p[m-1]=p[m]$ and u_g does not conclude (respectively, concludes) the algorithm. In this case, so (U_j), $1<j\leqq d$, cannot exist; otherwise Fact 8 would be contradicted. Thus, bound (17) holds for each

U_j, $1<j<d$, and (24) (respectively, (25)) holds for U_d. The total number of comparisons is obtained by adding these bounds to $m-c(u_1)$ (the contribution of u_1). The sum of the terms $m-c(u_1)$, $\text{line}_j-\text{line}_{j-1}$ and $s_1+c(s_1)-\text{line}_{d-1}$ is bounded by $s_1+c(s_1)-u_1-c(u_1)$. The sum of the terms $\lfloor(\text{last}(U_j)-\text{last}(U_{j-1}))(z'/2m)\rfloor$ and $\lfloor(\text{line}_{d-1}-\text{last}(U_{d-1}))((z'+1)/2m)\rfloor$ (respectively, $\lfloor(u_g-\text{last}(U_{d-1}))(z'/2m)\rfloor$) is bounded by $\lfloor(\text{last}(U_j)-u_1)((z'+1)/2m)\rfloor$ (respectively, $\lfloor(u_g-u_1)(z'/2m)\rfloor$). Therefore, (29) (respectively, (30)) holds.

Remaining cases. For each U_j, $1<j<d$, one of (17), (18), or (19) applies. Bound (19) is never larger than (18), since $\text{so}(U_j)-\text{last}(U_{j-1})\geqq z(\text{last}(U_{j-1})$ shifted the pattern by at least z since it matched a suffix of the pattern and the sequence of iterations from $\text{last}(U_{j-1})$ to $\text{so}(U_j)$ is increasing) and since $m=z+z'$, $z'<z$. Since $\text{so}(U_j)\leqq\text{last}(U_j)<\text{last}(U_{j+1})$, $1<j<d$, the sum of bounds (17) or (18) is bounded by $\text{line}_{d-1}-\text{line}_1+\lfloor(\text{last}(U_{d-1})-u_1)((z'+2)/2m)\rfloor$. Adding to this bound $m-c(u_1)$ and (26), we obtain (31) (since the bound of (26) is at least as large as the bounds of (24) and (25)).

We now show that iteration s_1 satisfies invariant 1 when it starts. Indeed, since u_1 satisfies Invariant 1, text positions in $[u_1+1,n]$ are either free or match all the noholes in $p[1,h_{\text{start}(u_1)}]$. Thus, text positions in $t[u_1+1,s_1]$ can become busy at the end of the heavy sequence. This amounts to s_1-u_1 text positions being declared busy. As for positions in $t[s_1+1,n]$, notice they are either free (they were free when u_1 started) or they match noholes in $p[1,h_{\text{start}(s_1)}]$. Therefore s_1 satisfies Invariant 1 when it starts.

Notice that the positions declared busy at the end of s_1 can be used to pay for s_1-u_1 character comparisons. This "covers" $s_1-u_1-c(u_1)$ comparisons performed by the algorithm during the heavy sequence as well as the $c(u_1)$ character comparisons (all matches) that u_1 has inherited from the previous iteration. The $c(s_1)$ matches inherited from u_g by s_1 are left unpaid. The remaining comparisons are paid by a credit line. □

LEMMA 20. *Consider a heavy sequence $u_1,u_2,\cdots,u_g$ and let s_1 be the iteration succeeding it. The number of character comparisons performed by Algorithm* SM′ *during the heavy sequence is bounded by*

$$s_1+c(s_1)-u_1-c(u_1)+\left\lfloor(u_g-u_1)\left(\min\left(\frac{1}{3},\frac{z'+2}{2m}\right)\right)\right\rfloor \tag{32}$$

when the pattern does not satisfy $p[1]=p[2]=p[m-1]=p[m]$ and u_g concludes the algorithm. It is bounded by

$$s_1+c(s_1)-u_1-c(u_1)+\left\lfloor(\text{line}_{d-1}-u_1)\left(\min\left(\frac{1}{3},\frac{z'+2}{2m}\right)\right)\right\rfloor \tag{33}$$

in all other cases.

Proof. Notice that (30) is obviously bounded by (32) since $z'/2m\leqq\frac{1}{4}$. We now prove (33). Let $\sum$ denote the alphabet. Let $\sum^{z+z'}$ denote the set of strings of length $m=z+z'$, where z is the period of the string and $z'<z$. Notice that (28), (29), and (31) in Lemma 19 partition $\sum^{z+z'}$ into three disjoint sets $\mathscr{A}=\{p[1]^l p[l+1]p[1]^l$: $l\geqq 2$ and $p[1]\neq p[l+1]\}$; $\mathscr{B}=\{p[1,m]$: not $(p[1]=p[m-1]=p[1]=p[m])\}$; $\mathscr{C}=\sum^{z+z'}-(\mathscr{A}\cup\mathscr{B})$. For each of the strings in these sets we prove that each of the corresponding bounds is bounded by (33).

Consider $\mathscr{A}$. Since $l\geqq 2$, $m=2l+1$ and $z'=l$ for this set of strings, (28) is bounded by (33).

Consider $\mathscr{B}$. We notice that $(z'+1)/2m \leqq (z'+2)/2m$. Therefore, in order to obtain (33) from (29), we need to show that $(z'+1)/2m \leqq \frac{1}{3}$. Such inequality follows easily on recalling that $m = z + z'$, $z' < z$ and $z \geqq 2$ for strings in $\mathscr{B}$.

Consider $\mathscr{C}$. In order to obtain (33) from (31), we first observe that for any string in $\mathscr{C}, m \geqq 6$ since, for $p \in \mathscr{C}, p[1] = p[2] = p[m-1] = p[m] (p \notin \mathscr{B})$ and $p \neq p[1]^l p[l+1] p[l]^l (p \notin \mathscr{A})$. (Recall the definition of $\mathscr{C}$.) We show that $(z'+2)/2m \leqq \frac{1}{3}$ for all of the strings in $\mathscr{C}$. Indeed, recalling that $m = z + z'$, $z' < z$, this inequality holds for any string in $\mathscr{C}$ having $z \geqq 5$. So assume $z < 5$. Since $p[1] = p[2] = p[m-1] = p[m]$, we have $z' \geqq 2$ and furthermore $z' = 3$ is not possible (for $z' = 3$, $z = 4$, $p = p[1]^3 p[4] p[1]^3 \in \mathscr{A}$). Thus, $z' = 2$ and since $m \geqq 6$, $(z'+2)/2m \leqq \frac{1}{3}$. □

6.2. Putting the sequences together again. We can finally prove the following theorem.

THEOREM 3. *Let $p[1, m]$ be a pattern with period z, $m = z + z'$ and $z' < z$, and let $t[1, n]$ be the text. Algorithm* SM′ *finds all occurrences of p in t in $O(n+m)$ time. In the worst case, the algorithm performs $n + \lfloor (n-m)(\min(\frac{1}{3}, (z'+2)/2m)) \rfloor$ character comparisons.*

Proof. The $O(n+m)$ time bound is obvious. As for the number of character comparisons, we notice that iteration zero satisfies Invariant 1 trivially and, by Lemmas 8 and 19, each iteration starting either a light or a heavy sequence satisfies the same invariant and the bounds in Lemmas 8 and 20 hold. The bound of the theorem follows by adding up these bounds: The sum of the terms $u_1 - s_1 + c(u_1) - c(s_1)$ or $s_1 - u_1 + c(s_1) - c(u_1)$ is bounded by n, and the sum of the terms $\lfloor (\text{line}_{d-1} - u_1)(\min(\frac{1}{3}, (z'+2)/2m)) \rfloor$ or $\lfloor (u_g - u_1)(\min(\frac{1}{3}, (z'+2)/2m)) \rfloor$ is bounded by $\lfloor (n-m)(\min(\frac{1}{3}, (z'+2)/2m)) \rfloor$. Since $c(u) = 0$ for u the dummy iteration, no comparison is left unpaid. □

The bound in Theorem 3 is tight for $m = 3$ and any $n = 3c$, c any integer. Indeed, let $p[1, m] = aba$ and $t[1, n] = (aba)^c$. By Fact 8 and the definition of a light sequence, there is no iteration u of SM′ such that u handles a shift by one and $u < t\text{last}(u) - 1$. Therefore, SM′ simulates SM when searching for aba in any text. In the previous section we have shown that SM performs $n + \lfloor (n-m)/3 \rfloor$ character comparisons when it searches for all occurrences of aba in $t[1, n] = (aba)^c$. Since for the chosen pattern $z' = 1$ and $m = 3$, this bound is equal to the one in Theorem 3.

7. Concluding remarks. We have shown that, for any pattern of length m and any text of length n, $c(n, m) \leqq c_{\text{on-line}}(n, m) \leqq n + \lfloor (n-m)(\min(\frac{1}{3}, (z'_s+2)/2(z_s + z'_s))) \rfloor \leqq \frac{4}{3}n - \frac{1}{3}m$ character comparisons, where (z_s, z'_s, k_s) is the last term of the periodic decomposition of the pattern defined in the Introduction. In a companion paper [11], we show a lower bound for on-line algorithms that is equal to $\frac{4}{3}n - \frac{1}{3}m$ for $m = 3$. Our algorithm is based on a new analysis of the string matching algorithm by Colussi [7]. Moreover, our analysis of Colussi's algorithm confirms the experimental results showing that it performs very well in practice.

Acknowledgments. We praise the endurance of the referee, who survived this tour de force in analysis of algorithms with enough energy left to give us very helpful comments.

REFERENCES

[1] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] A. APOSTOLICO AND M. CROCHEMORE, *Optimal canonization of all substrings of a string*, Tech. Report, TR 89-75, LITP, Université de Paris 7, Paris, France, October, 1989.

[3] A. Apostolico and R. Giancarlo, *The Boyer-Moore-Galil string searching strategies revisited.* SIAM J. Comput., 15 (1986), pp. 98-105.
[4] S. W. Bent and J. W. John, *Finding the median requires 2n comparisons*, in Proc. 17th Symposium on Theory of Computing, Association for Computing Machinery, 1985, pp. 213-216.
[5] R. S. Boyer and J. S. Moore, *A fast string searching algorithm*, Comm. ACM, 20 (1977), pp. 762-772.
[6] R. Cole, *Tight bounds on the complexity of the Boyer-Moore string matching algorithm*, in Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1991.
[7] L. Colussi, *Correctness and efficiency of string matching algorithms*, Inform. and Comput., to appear.
[8] M. Crochemore and D. Perrin, *Two way pattern matching*, Tech. Report, Dept. of Computer Science, Université de Paris, Paris, France, 1989.
[9] L. R. Ford and S. M. Johnson, *A tournament problem*, Amer. Math. Monthly, 66 (1959), pp. 387-389.
[10] Z. Galil, *On improving the worst case running time of the Boyer-Moore string matching algorithm*, Comm. ACM, 22 (1979), pp. 505-508.
[11] Z. Galil and R. Giancarlo, *On the exact complexity of string matching: Lower bounds*, SIAM J. Comput., 20 (1991), pp. 1008-1020.
[12] R. Giancarlo, *Efficient Algorithms on Strings*, Ph.D. thesis, Dept. of Computer Science, Columbia University, New York, 1990.
[13] L. J. Guibas and A. M. Odlyzko, *A new proof of the linearity of the Boyer-Moore string matching algorithm*, SIAM J. Comput., 9 (1980), pp. 672-682.
[14] D. G. Kirkpatrick, *Topics in the complexity of combinatorial algorithms*, Tech. Report, Dept. of Computer Science, University of Toronto, Toronto, Canada, 1974.
[15] ———, *A unified lower bound for selection and set partitioning problems*, J. ACM, 28 (1981), pp. 150-165.
[16] S. S. Kislitsyn, *On the selection of the kth element of an ordered set by pairwise comparison*, Sibirsk. Mat. Zh., 5 (1964), pp. 557-564.
[17] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.
[18] D. E. Knuth, J. H. Morris, and V. B. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 189-195.
[19] R. C. Lyndon and M. P. Schutzenberger, *The equation* $a^m = b^n c^p$ *in a free group*, Michigan Math. J., 9 (1962), pp. 289-298.
[20] I. Pohl, *A sorting problem and its complexity*, Comm. ACM, 15 (1972), pp. 462-464.
[21] A. Schonhage, M. Paterson, and N. Pippenger, *Finding the median*, J. Comput. System Sci., 13 (1976), pp. 184-199.
[22] J. Schreier, *On tournament elimination systems*, Mathesis Polska, 7 (1932), pp. 154-160.
[23] C. K. Yap, *New upper bounds for selection*, Comm. ACM, 19 (1979), pp. 501-508.

SIAM J. COMPUT.
Vol. 21, No. 3, pp. 438-449, June 1992

© 1992 Society for Industrial and Applied Mathematics
002

PROBABLY APPROXIMATE LEARNING OVER CLASSES OF DISTRIBUTIONS*

B. K. NATARAJAN†

Abstract. This paper presents algorithms that construct approximations to sets and functions on the reals, from randomly chosen sample points. The model that is analyzed is a generalization of the paradigm of *probably approximate learning* proposed by Valiant. Previously, necessary and sufficient conditions for learning sets were established for the case when the class of sampling distributions is finite, and for the case when the class of sampling distributions is the set of all possible distributions. Here, sufficient conditions are obtained for learning sets and functions over general classes of sampling distributions. The results for functions are with respect to general metrics for measuring the distance between two functions.

Key words. probabilistic learning, sets, functions, classes of distributions

AMS(MOS) subject classifications. 68T05, 62G99

1. Introduction. Valiant [17] introduced a formal framework for the probabilistic analysis of algorithms that learn sets defined on a predetermined universe. The framework requires the learning algorithm to construct an approximation to a set, given some randomly chosen examples of members and nonmembers of the set. The examples are chosen at random according to an arbitrary but unknown probability distribution on the universe. It is sufficient if the algorithm is "probably approximate" in that it need only construct an approximation to the set with high probability. Blumer et al. [4] obtain necessary and sufficient conditions for such learning, using the methods of Vapnik and Chervonenkis [20] on the probabilistic convergence of classes of events. Natarajan [12] obtains corresponding conditions for countable universes, using simple counting arguments. Numerous authors report related results (see [1], [9], [15], [17], amongst others). Also see [13] for an introductory overview.

As a natural extension of the above results, we seek to obtain the necessary and sufficient conditions for learning when the sampling distribution is one of a fixed and known class of distributions. Benedek and Itai [3] obtained such conditions for the case when the class of sampling distributions is of finite cardinality. Our result holds for classes of finite or infinite cardinality. In essence, they involve a sharpening of some of the results of Vapnik and Chervonenkis [20], in that we obtain an estimate of the rate of convergence, whereas [20] proves convergence in the limit only.

In [12] and [14], the notion of probably approximate learning is extended to functions on the reals, presenting necessary and sufficient conditions for learning over all probability distributions, with respect to the discrete metric on function spaces. Here, we present sufficient conditions for the learnability of functions with respect to an arbitrary metric and class of sampling distributions. Additional results on this problem may be found in [8].

The results presented here appeared in preliminary form in [10] and [11].

2. Learning concepts. Let $\mathbf{R}$ be the set of reals. A set[1] $f \subseteq \mathbf{R}$ is called a *concept.* A set of concepts $F \subseteq 2^{\mathbf{R}}$ is called a *concept class.* The indicator function $I_f : \mathbf{R} \to \{0, 1\}$ of

* Received by the editors June 21, 1989; accepted for publication (in revised form) June 26, 1991.

† Hewlett-Packard Laboratories, 1501 Page Mill Road, Building 3U., Palo Alto, California 94304.

[1] All sets are assumed to be Borel [5].

a concept f is such that $I_f(x)=1$ if $x \in f$ and $I_f(x)=0$ otherwise. In the interest of simplicity, we use f to denote both the concept f and its indicator function I_f, depending on the context for clarity. An *example* for a concept $f \subseteq \mathbf{R}$, is a pair $(x, f(x))$, where $x \in \mathbf{R}$. We say that a learning algorithm A learns F if it can construct an approximation to every $f \in F$, from randomly chosen examples for f.

In the above, we use the term "algorithm" in a nontraditional sense: we are interested mainly in the function computed by the learning algorithm, but we use the term "algorithm" as it is often convenient to express the function in a procedural form. In particular, the term algorithm is used to refer to any procedure that computes a random function [6], i.e., the probability that the algorithm outputs a certain string on a certain input is defined. Additional discussion on this characterization of learning algorithms may be found in [12], [13].

The concept that is to be learned is called the *target concept.* To aid it in its task, the learning algorithm is provided with a subroutine EXAMPLE, which at each call returns an example for the target concept f. The EXAMPLES are chosen at random according to a probability distribution P on $\mathbf{R}$. Specifically, for any set $S \subseteq \mathbf{R}$, with probability $P(S)$, a call of EXAMPLE will produce an example $(x, f(x))$, for some $x \in S$.

Notation. For a set $S \subseteq \mathbf{R}$, $P(S)$ is the weight of P on S, i.e., $P(S)=\int_S dP$.

We say F is a well-behaved class if it satisfies the measurability conditions stated in [4]. Henceforth we will be concerned only with well-behaved classes.

We now give a formal definition of learnability. In words, we say that a class of concepts F is learnable if there exists a learning algorithm than can construct arbitrarily good approximations to the concepts in F from finitely many examples. The number of examples required by the algorithm may depend on the probability distribution P and on the precision parameters. Previously, Ben-David et al. [2] examined some aspects of this model of learning under the name "learnability nonuniform in the distribution."

DEFINITION. A class F is *learnable* over a class of distribution Ξ if there exists a learning algorithm A such that

(1) A takes as input reals ε, $\delta \in (0, 1]$, where ε is the error parameter and δ is the confidence parameter.

(2) A may call EXAMPLE. EXAMPLE returns examples for some f in F, where the examples are chosen randomly according to some distribution $P \in \Xi$. The number of calls of EXAMPLE must be finite, although the exact number may depend on ε, δ, and P.

(3) For all probability distributions $P \in \Xi$ and all $f \in F$, with probability at least (1-δ), A outputs $g \in F$ such that $P(f \Delta g) \leqq \varepsilon$.

Notation. For two sets f and g, $f \Delta g$ is the symmetric difference between f and g, i.e., $f \Delta g=(f-g) \cup (g-f)$.

In the above definition, the number of examples drawn by A may depend on the probability distribution P. If the number of examples drawn by A depends only on ε and δ, we say F is *uniformly learnable* over Ξ.

DEFINITION. Let $s^l=(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$ be a sequence of examples. A concept f is said to be *consistent* with s^l if for each (x_i, y_i) in s^l, $y_i=f(x_i)$.

A consistent algorithm for a class F is one that draws a certain number of examples and then simply outputs a concept in F that is consistent with all the examples obtained.

DEFINITION. A is a *consistent* algorithm for a class F if

(1) A takes as input ε, $\delta \in (0, 1]$.

(2) A may call EXAMPLE.

(3) A outputs $g \in F$ consistent with all the examples obtained during its execution.

Let $\mathbf{N}$ be the natural numbers and let $m:(0,1]\times(0,1]\rightarrow\mathbf{N}$. Note that since $\mathbf{N}$ is the range of m, $m(\varepsilon,\delta)$ is finite for all ε and δ in $(0,1]$. Learnability by consistent algorithms is called "solid learnability" [2].

DEFINITION. A class of concepts F is *solidly learnable* over a class of distributions Ξ if there exists $m:(0,1]\times(0,1]\rightarrow\mathbf{N}$ such that every consistent algorithm that calls for at least $m(\varepsilon,\delta)$ examples on input ε, δ is a learning algorithm for F.

Notice that solid learnability implies uniform learnability, which in turn implies learnability.

2.1. Previous results. In [4], conditions necessary and sufficient for solid learnability are obtained for the case where Ξ is the class of all distributions on $\mathbf{R}$. To be precise, they obtain their results for a slightly different formulation of the learning paradigm, but one that is equivalent to ours as reasoned in [7]. Blumer et al. [4] use the methods of Vapnik and Chervonenkis [20], as well as the notion of shattering defined below.

DEFINITION. Let F be a class of concepts and let $S\subseteq\mathbf{R}$. *F shatters S* if the set $\{f\cap S\mid f\in F\}$ is the power set of S.

DEFINITION. The *Vapnik–Chervonenkis dimension* of F is the least integer d such that every set shattered by F is of cardinality at most d. We denote the Vapnik-Chervonenkis dimension of F by $\mathbf{D}_{\mathrm{VC}}(F)$.

THEOREM 1 ([20], [4]). *A concept class F is solidly learnable over the class of all distributions if and only if $\mathbf{D}_{\mathrm{VC}}(F)$ is finite.*

Benedek and Itai [3] present conditions necessary and sufficient for uniform learnability over a finite class of distributions. They define the notion of an ε-cover of a class F as follows.

DEFINITION. For concept class F, distribution P, and $\varepsilon\in(0,1]$, an ε-cover of F is a subset G of F such that for all $f\in F$, there exists $g\in G$ such that $P(f\Delta g)\leqq\varepsilon$.

THEOREM 2 ([3]). *Let F be a concept class and Ξ a class of distributions of finite cardinality. F is uniformly learnable over Ξ if and only if for all $\varepsilon\in(0,1]$ and each $P\in\Xi$, there exists a finite ε-cover of F.*

2.2. Learning concepts over infinite classes of distributions. We now obtain sufficient conditions for the learnability of concepts over a possibly infinite class of distributions. Our results will lean heavily on certain convergence results of [20]. Also see [16], [19], and [21] for related results. We need the following definitions.

DEFINITION. Let P be a probability distribution on $\mathbf{R}$. An *l-sample x^l* (of $\mathbf{R}$ with respect to P) is the sequence of l reals obtained by l random draws from $\mathbf{R}$ according to P.

Let $f\subseteq\mathbf{R}$ and let x^l be an l-sample. We use $f\cap x^l$ to denote the sequence of those items in x^l that are also elements of f. That is, $f\cap x^l$ is the sequence $y_1, y_2, \cdots, y_k$, where y_i is the ith item of x^l that is also an element of f. We call $f\cap x^l$ the *subsample* of x^l *induced* by f. For instance, let $f=\{7, 3, 33.4\}$ and let $x^l=3, 3, 4.5, 7, 17$. Then, $f\cap x^l$ is the sequence 3, 3, 7. We use $F\cap x^l$ to denote the set of distinct subsamples of x^l induced by all the concepts in F.

DEFINITION. For concept class F, probability distribution P, and sample size l, the *entropy* $H_{P,l}(F)$ is given by

$$H_{P,l}(F)=\mathbf{E}\{\log|F\cap x^l|\},$$

where x^l is an l-sample drawn according to P.

Notation. We use "log" to denote $\log_2$ and "ln" to denote $\log_e$. For a random variable r, $\mathbf{E}(r)$ and $\mathbf{V}(r)$ denote the expectation and variance of r, respectively. For a set S, $|S|$ denotes the cardinality of S.

We now state our main result.

THEOREM 3. *A class of concepts F is solidly learnable over a class of distributions Ξ if*

$$\limsup_{l\to\infty} \sup_{P\in\Xi} \frac{H_{P,l}(F)}{l}=0.$$

Proof. We will prove that there exists a function m such that any consistent algorithm drawing at least $l=m(\varepsilon,\delta)$ examples is a learning algorithm for F. The proof is essentially a sharpening of the results of Vapnik and Chervonenkis [20], in the sense that we estimate the rate of convergence where they prove convergence in the limit. Our proof uses some of the intermediate results of [20].

For event E, let $\mathbf{Pr}\{E\}$ denote the probability of E occurring. Let x^l be an l sample and let $f\subseteq \mathbf{R}$. The hit frequency $\nu(x^l,f)$ is the fraction of items in x^l that are elements of f, i.e.,

$$\nu(x^l,f)=\frac{|f\cap x^l|}{|x^l|}. \qquad \square$$

Notation. For a sequence x^l, $|x^l|$ denotes the number of items in x^l.

Let $\eta\in(0,1]$. For two independently drawn l-samples x^l and y^l, define the two events Q and C as follows:

$$Q=\left\{\left(\sup_{g\in F}|\nu(x^l,g)-P(g)|\right)>\eta\right\},$$

$$C=\left\{\left(\sup_{g\in F}|\nu(x^l,g)-\nu(y^l,g)|\right)>\eta\right\}.$$

Lemma 1 of [20] (or Claim 4.1 of [13]) states that for

$$l>\frac{2}{\eta^2}, \tag{1}$$

$\mathbf{Pr}\{C\}\geqq\frac{1}{2}\mathbf{Pr}\{Q\}$. We will now estimate $\mathbf{Pr}\{C\}$.

Let $x^l=x_1,x_2,\cdots,x_l$ be an l-sample. Define ζ^l to be the random variable given by

$$\zeta^l=\frac{1}{l}\log|F\cap x^l|.$$

Using symmetrization arguments, Vapnik and Chervonenkis [20, p. 276] show that

$$\mathbf{Pr}\{C\}\leqq 2\left(\frac{2}{e}\right)^{\eta^2 l/8}+\mathbf{Pr}\left\{\zeta^{2l}>\frac{\eta^2}{16}\right\}. \tag{2}$$

We now estimate the right-hand side of the above inequality.

Let $q:(0,1]\to\mathbf{N}$ be such that for $\mu\in(0,1]$, for all $l\geqq q(\mu)$

$$\sup_{P\in\Xi}\frac{H_{P,l}(F)}{l}\leqq\mu.$$

Such a function q must exist because

$$\limsup_{l\to\infty}\sup_{P\in\Xi}\frac{H_{P,l}(F)}{l}=0.$$

Since $0 \leqq \zeta^{2l} \leqq 1$, $\mathbf{V}(\zeta^{2l}) \leqq \mathbf{E}(\zeta^{2l})$. Using this and Chebyshev's inequality, we can write

$$\mathbf{Pr}\{|\zeta^{2l} - \mathbf{E}(\zeta^{2l})| > \mu\} \leqq \frac{\mathbf{E}(\zeta^{2l})}{\mu^2}.$$

This implies that

$$\mathbf{Pr}\{\zeta^{2l} > \mathbf{E}(\zeta^{2l}) + \mu\} \leqq \frac{\mathbf{E}(\zeta^{2l})}{\mu^2}.$$

By the definition of q, if

$$l \geqq q(\mu^3)/2, \tag{3}$$

$\mathbf{E}(\zeta^{2l}) \leqq \mu^3$. For such l,

$$\mathbf{Pr}\{\zeta^{2l} > \mu^3 + \mu\} \leqq \mathbf{Pr}\{\zeta^{2l} > \mathbf{E}(\zeta^{2l}) + \mu\} \leqq \frac{\mathbf{E}(\zeta^{2l})}{\mu^2} \leqq \mu.$$

Choosing

$$\mu^3 + \mu \leqq \eta^2/16, \tag{4}$$

we have

$$\mathbf{Pr}\{\zeta^{2l} > \eta^2/16\} \leqq \mu.$$

We would like to pick l so that $\mathbf{Pr}\{Q\} \leqq 2\mathbf{Pr}\{C\} \leqq \delta$. To do so, it suffices to pick l so that each term on the right-hand side of (2) is at most $\delta/4$. Specifically,

$$4(2/e)^{\eta^2 l/8} \leqq \frac{\delta}{4},$$

and

$$\mathbf{Pr}\{\zeta^{2l} > \eta^2/16\} \leqq \mu \leqq \frac{\delta}{4}.$$

After some manipulation, the former yields

$$l \geqq \frac{8}{\eta^2}\left(\frac{4\ln(2) - \ln(\delta)}{1 - \ln(2)}\right). \tag{5}$$

The latter yields the additional constraint on μ,

$$\mu \leqq \frac{\delta}{4}. \tag{6}$$

Let f be the target concept, and let s^l be a sequence of l randomly chosen examples for f. Let $G \subseteq F$ be given by $G = \{g \mid P(f \Delta g) > \varepsilon\}$. We wish to estimate the probability that some $g \in G$ agrees with all of these examples, i.e., $\mathbf{Pr}\{g \in G$ is consistent with $s^l\}$.

Let $s^l = (x_1, y_1), (x_2, y_2), \cdots, (x_l, y_l)$. Consider the sample $x^l = x_1, x_2, x_3, \cdots, x_l$ of $\mathbf{R}$, obtained by discarding the labels $y_1, y_2, \cdots, y_l$ from s^l. Now,

$$\mathbf{Pr}\{g \in G \text{ is consistent with } s^l\} \leqq \mathbf{Pr}\{\nu(x^l, g) = \nu(x^l, f) \text{ and } g \in G\}.$$

Since $g \in G$, $|P(g) - P(f)| > \varepsilon$. Therefore, if $\nu(x^l, g) = \nu(x^l, f)$, at least one of the following inequalities must be true:

$$|\nu(x^l, g) - P(g)| > \frac{\varepsilon}{2}, \qquad |\nu(x^l, f) - P(f)| > \frac{\varepsilon}{2}.$$

Hence,

$$\begin{aligned}\mathbf{Pr}\,\{g \in G \text{ is consistent with } s^l\} &\leqq \mathbf{Pr}\,\{\nu(x^l, g) = \nu(x^l, f) \text{ and } g \in G\}\\ &\leqq \mathbf{Pr}\left\{\left(\sup_{g \in G} |\nu(x^l, g) - P(g)|\right) > \varepsilon/2\right\}\\ &\quad + \mathbf{Pr}\,\{|\nu(x^l, f) - P(f)| > \varepsilon/2\}\\ &\leqq 2\mathbf{Pr}\left\{\left(\sup_{g \in F} |\nu(x^l, g) - P(g)|\right) > \varepsilon/2\right\}.\end{aligned}$$

The above is exactly $2\mathbf{Pr}\,\{Q\}$ for $\eta = \varepsilon/2$. Thus, if we set $\eta = \varepsilon/2$, and μ satisfies inequalities (4) and (6), and l satisfies inequalities (1), (3), and (5), we have ensured that

$$\mathbf{Pr}\,\{g \in G \text{ is consistent with } s^l\} \leqq 2\mathbf{Pr}\left\{\left(\sup_{g \in F} |\nu(x^l, g) - P(g)|\right) > \varepsilon/2\right\} \leqq \delta.$$

By manipulating the inequalities listed above, we can now show that it suffices for l to satisfy

$$l \geqq q(\mu^3) + \frac{64}{\varepsilon^2}\left(4 + \ln\left(\frac{1}{\delta}\right)\right),$$

where $\mu = \min\{\delta/4, \varepsilon^2/128\}$.

Thus, there exists $l = m(\varepsilon, \delta)$ so that, if at least l random examples are drawn, any concept $g \in F$ consistent with these examples will, with probability at least $(1-\delta)$, be such that $P(f \Delta g) \leqq \varepsilon$. It follows that F is solidly learnable. □

We will now show that the conditions of the above theorem are sufficient but not necessary. Specifically, we exhibit a concept class F and a class of distributions Ξ such that F is solidly learnable over Ξ and yet does not satisfy the conditions of Theorem 3.

Example 1. Let Ξ consist of the single distribution P that is uniform on the interval $[0, 1]$. Let F be the set of all finite subsets of $[0, 1]$. For any $\varepsilon \in (0, 1]$, and any pair $f, g \in F$, $P(f \Delta g) < \varepsilon$. It follows that any consistent algorithm that draws at least say, one, example is trivially a learning algorithm for F. Hence F is solidly learnable over Ξ.

For finite l with probability 1, an l-sample x^l will have l distinct items. Thus, with probability 1, $|F \cap x^l| = 2^l$. Hence, $\lim_{l \to \infty} H_{P,l}(F)/l = 1$. Thus, the conditions of Theorem 3 are violated.

We now identify conditions necessary for uniform learnability over a class of distributions.

THEOREM 4. *F is uniformly learnable over Ξ only if for all $\varepsilon \in (0, 1]$, there exists finite integer N_ε such that with respect to every distribution in Ξ, there exists an ε-cover of F of cardinality at most N_ε.*

Proof. This is essentially the "only if" direction of Theorem 2 and may be proved as in [3]. □

3. Learning functions. In the foregoing, we were concerned with learning approximations to concepts or sets. In the more general setting, we may consider learning functions from $\mathbf{R}$ to $\mathbf{R}$. To do so, we generalize our formulation of the problem.

Consider a function $f: \mathbf{R} \to \mathbf{R}$. An example for f is a pair $(x, f(x))$, $x \in \mathbf{R}$. The set graph (f) is the set of all examples for f, i.e.,

$$\text{graph}\,(f) = \{(x, y) \mid x \in \mathbf{R}, y = f(x)\}.$$

We restrict our discussion to those functions whose graphs are Borel sets. A class of functions F is any set of such functions from $\mathbf{R}$ to $\mathbf{R}$.

A learning algorithm for a class of functions F is an algorithm that attempts to infer approximations to functions in F from examples for it. The learning algorithm has at its disposal a subroutine EXAMPLE, which at each call produces a randomly chosen example for the target function f. The examples are chosen according to an arbitrary and unknown probability distribution P on $\mathbf{R}$,

In order to make precise the notion of two functions being approximately equal, we consider metrics on the space of functions as follows. We will measure the "distance" between two functions to be the expected distance between their values, measured in some metric on $\mathbf{R}$. Formally, a metric on $\mathbf{R}$ is a function $L:\mathbf{R}\times\mathbf{R}\to\mathbf{R}$ such that for any $x, y, z\in\mathbf{R}$: (a) $L(x, y)\geqq 0$; (b) $L(x, x)=0$; (c) $L(x, y)=L(y, x)$; and (d) $L(x, y)+L(y, z)\geqq L(x, z)$. For instance, consider the Euclidean metric on $\mathbf{R}$, $L(x, y)=((x-y)^2)^{1/2}=|x-y|$. The corresponding distance between two functions f and g with respect to a distribution P would be $\int_{x\in\mathbf{R}}|f(x)-g(x)|\,dP$.

We can now define a notion of learnability in this setting.

DEFINITION. A class of functions F is *learnable* over a class of distributions Ξ with respect to a metric L if there exists an algorithm A such that

(1) A takes as input ε, $\delta\in(0, 1]$.

(2) A may call EXAMPLE. EXAMPLE returns examples for some function f in F, where the examples are chosen randomly according to some distribution $P\in\Xi$. The number of calls of EXAMPLE must be finite, although the exact number may depend on ε, δ, and P.

(3) For all probability distributions $P\in\Xi$ and all functions f in F, with probability at least $(1-\delta)$, A outputs $g\in F$ such that

$$\int_{x\in\mathbf{R}} L(f(x), g(x))\,dP\leqq\varepsilon.$$

The notions of uniform learnability and solid learnability carry over to this setting without change.

For the case of the *discrete metric* defined by

$$L_d(x, y)=\begin{cases}0 & \text{if } x=y,\\ 1 & \text{otherwise,}\end{cases}$$

and for Ξ being the set of all distributions, [12] presents conditions necessary and sufficient for learnability. As we will be using these results, we briefly review them below.

For a class of functions F,

$$\text{graph}\,(F)=\{\text{graph}\,(f)\,|\,f\in F\}.$$

The following is a generalization of the notion of shattering.

DEFINITION. Let F be a class of functions from a set X to Y. We say F *shatters* a set $S\subseteq X$ if there exist two functions f, $g\in F$ such that

(1) for all $x\in S$, $f(x)\neq g(x)$.

(2) for all $S_1\subseteq S$, there exists $h\in F$ such that h agrees with f on S_1 and with g on $S-S_1$, i.e., for all $x\in S_1$, $h(x)=f(x)$, for all $x\in S-S_1$, $h(x)=g(x)$.

DEFINITION. Let F be a class of functions from $\mathbf{R}$ to $\mathbf{R}$. The *generalized dimension* of F is the least integer d such that every set shattered by F is of cardinality at most d. We denote the generalized dimension by $\mathbf{D}_G(F)$.

THEOREM 5. *Let F be a class of functions from* $\mathbf{R}$ *to* $\mathbf{R}$.

(1) *F is solidly learnable over the class of all probability distributions with respect to the discrete metric if* $\mathbf{D}_{\mathrm{VC}}(\mathrm{graph}\,(F))$ *is finite.*[2]

(2) *F is learnable over the class of all distributions with respect to the discrete metric only if* $\mathbf{D}_{\mathrm{G}}(F)$ *is finite.*

It can be shown that $\mathbf{D}_{\mathrm{VC}}(\mathrm{graph}\,(F)) \geqq \mathbf{D}_{\mathrm{G}}(F)$. Also, $\mathbf{D}_{\mathrm{G}}(F)$ may be finite while $\mathbf{D}_{\mathrm{VC}}(\mathrm{graph}\,(F))$ is infinite, leaving a gap between the "if" and "only if" conditions of the above theorem. This ends our review of results from [12].

We now use the above results to establish conditions sufficient for learning with respect to an arbitrary metric. To do so, we need the following definition. The diameter of a class of functions F with respect to a metric L is the greatest value attained by $L(f(x), g(x))$ over every pair of functions $f, g \in F$ and every point $x \in \mathbf{R}$.

DEFINITION. Let F be a class of functions from $\mathbf{R}$ to $\mathbf{R}$ and let L be a metric on $\mathbf{R}$. The *diameter* $D_L(F)$ of F with respect to L is given by

$$D_L(F) = \sup_{x \in \mathbf{R}, f, g \in F} L(f(x), g(x)).$$

We can now state the following corollary to Theorem 5.

COROLLARY 1. *Let F be a class of functions from* $\mathbf{R}$ *to* $\mathbf{R}$ *such that* $\mathbf{D}_{\mathrm{VC}}(\mathrm{graph}\,(F))$ *is finite, and L be a metric on* $\mathbf{R}$ *such that* $\rho = D_L(F)$ *is finite. Then, F is solidly learnable over the class of all probability distributions with respect to L.*

Proof. Since $D_L(F) = \rho$, for all $x \in \mathbf{R}$, $L(f(x), g(x)) \leqq \rho$. Thus,

$$\int_{x \in \mathbf{R}} L(f(x), g(x))\, dP = \int_{f(x) \neq g(x)} L(f(x), g(x))\, dP \leqq \rho \int_{f(x) \neq g(x)} dP.$$

$$= \rho \int_{x \in \mathbf{R}} L_d(f(x), g(x))\, dP.$$

It follows that if $\int_{x \in \mathbf{R}} L_d(f(x), g(x))\, dP \leqq \varepsilon/\rho$, then $\int_{x \in \mathbf{R}} L(f(x), g(x))\, dP \leqq \varepsilon$.

Since $\mathbf{D}_{\mathrm{VC}}(\mathrm{graph}\,(F))$ is finite, Theorem 5 implies that F is solidly learnable with respect to the discrete metric. That is, there exist $m : (0, 1] \times (0, 1] \to \mathbf{N}$ such that any consistent algorithm that draws $m(\varepsilon, \delta)$ examples is a learning algorithm for F with respect to the discrete metric. Let f be the target function. Thus, any consistent algorithm that draws at least $m(\varepsilon/\rho, \delta)$ examples will output a function g such that with probability at least $(1-\delta)$, $\int_{x \in \mathbf{R}} L_d(f(x), g(x))\, dP \leqq \varepsilon/\rho$. It follows that any consistent algorithm that draws at least $m(\varepsilon/\rho, \delta)$ examples will output a function g such that with probability at least $(1-\delta)$, $\int_{x \in \mathbf{R}} L(f(x), g(x))\, dP \leqq \varepsilon$. Hence the result. □

We now consider the case when Ξ is a finite class of distributions on $\mathbf{R}$. To do so, we generalize the notion of an ε-cover as follows.

DEFINITION. An ε-cover for a class of functions F with respect to metric L and distribution P is a class of functions $G \subseteq F$ such that for every $f \in F$, there exists $g \in G$ satisfying

$$\int_{x \in \mathbf{R}} L(f(x), g(x))\, dP \leqq \varepsilon.$$

We can now state the following result.

THEOREM 6. *Let F be class of functions, and L a metric such that* $\rho = D_L(F)$ *is finite. F is uniformly learnable with respect to L over a finite class of distributions* Ξ, *if for all* $\varepsilon \in (0, 1]$ *and* $P \in \Xi$, *there exists a finite* ε*-cover of F.*

[2] Again, graph (F) is assumed to be well behaved, as in [4].

Proof. The following is a learning algorithm for F. Notice that the number of examples drawn by the algorithm does not vary with the distribution.

Learning Algorithm A_1
input: ε, $\delta \in (0, 1]$.
begin
Let $\{C_1, C_2, \cdots, C_k\}$
be finite $(\varepsilon/2)$-covers for F
with respect to the distributions
$\{P_1, P_2, \cdots, P_k\}$ of Ξ.
Let $C \bigcup_{i=1}^{k} C_i$.
Pick $l \geqq |C| \dfrac{4\rho^2}{\delta\varepsilon^2}$.
Call for l examples.
Let $(x_1, y_1), (x_2, y_2), \cdots, (x_l, y_l)$
be the sequence of examples obtained.
if there exists $c \in C$ such that

$$\sum_{i=1}^{l} L(y_i, c(x_i)) \leqq \frac{\varepsilon}{2}$$

then output c.
else output any $c \in C$.
end

On input ε, δ the algorithm constructs finite $(\varepsilon/2)$-covers for F with respect to each distribution in Ξ. Since Ξ is finite, the union C of these covers is also finite.

Let f be the target function, let x^l be the l-sample $x_1, x_2, \cdots, x_l$ and let $c \in C$. Consider the random variable ϕ defined by

$$\phi = \frac{1}{l} \sum_{i=1}^{l} L(y_i, c(x_i)) = \frac{1}{l} \sum_{i=1}^{l} L(f(x_i), c(x_i)).$$

The variance of ϕ can be written as

$$\mathbf{V}(\phi) = \mathbf{V}\left(\frac{1}{l} \sum_{i=1}^{l} L(f(x_i), c(x_i))\right) = \frac{1}{l^2} \mathbf{V}\left(\sum_{i=1}^{l} L(f(x_i), c(x_i))\right).$$

Since the x_i are mutually independent, we have

$$\mathbf{V}(\phi) = \frac{1}{l^2} \sum_{i=1}^{l} \mathbf{V}(L(f(x_i), c(x_i))). \tag{7}$$

Surely,

$$\mathbf{V}(L(f(x_i), c(x_i))) \leqq \rho^2.$$

Combining the above with (10), we get, $\mathbf{V}(\phi) \leqq \rho^2/l$.

Invoking Chebyshev's inequality, we get

$$\mathbf{Pr}\left(|\phi - \mathbf{E}(\phi)| > \frac{\varepsilon}{2}\right) \leqq \frac{4\rho^2}{l\varepsilon^2}.$$

By definition, $\mathbf{E}(\phi) = \int_{x \in \mathbf{R}} L(f(x), c(x))\, dP$. Hence,

$$\mathbf{Pr}\left(\left|\phi - \int_{x \in \mathbf{R}} L(f(x), c(x))\, dP\right| > \frac{\varepsilon}{2}\right) \leqq \frac{4\rho^2}{l\varepsilon^2}.$$

It follows that

$$\mathbf{Pr}\left(\sup_{c\in C}\left|\phi-\int_{x\in\mathbf{R}} L(f(x),c(x))\,dP\right|>\frac{\varepsilon}{2}\right)\leqq|C|\frac{4\rho^2}{l\varepsilon^2}.$$

Thus, if l is such that $|C|4\rho^2/l\varepsilon^2\leqq\delta$, then, with probability at least $(1-\delta)$, any function $c\in C$ for which $\sum_{i=1}^{l} L(f(x_i),c(x_i))\leqq\varepsilon/2$ must satisfy $\int_{x\in\mathbf{R}} L(f(x),c(x))\,dP\leqq\varepsilon$.

Since C contains an $\varepsilon/2$-cover for F with respect to P, it follows that with probability at least $(1-\delta)$, the algorithm will find such c. This completes the proof. □

Unlike Theorem 2, Theorem 6 is not tight. This is illustrated below.

Example 2. Let $h\subseteq\mathbf{R}$ be the union of finitely many closed intervals with rational endpoints in $[\frac{1}{2},1]$. That is, $h=\bigcup_{i=1}^{n}[a_i,b_i]$, where a_i and b_i are rationals in $[\frac{1}{2},1]$.

Define the function f_h as follows.

$$f_h(x)=\begin{cases}.\alpha & \text{if } x\in[0,\frac{1}{2}],\\ h(x) & x\in[\frac{1}{2},1],\\ 0 & \text{otherwise},\end{cases}$$

where α is an encoding of the intervals of h, and $.\alpha$ simply places the decimal point ahead of α to keep $|f(x)|\leqq 1$. (Recall that $h(x)$ is the value of the indicator function $I_h(x)$.) For instance, say $h=[\frac{1}{4},\frac{1}{3}]\cup[\frac{2}{3},\frac{3}{4}]$. Then $\alpha=101^4 00101^3 001^2 01^3 1^3 01^4$, where 1^a is the string of a 1's. That is, α encodes the endpoints of h in unary, separating them with zeros.

Now, let F be the class of functions defined as above, over all such sets h. Let P be the uniform distribution on the closed interval $[0,1]$. With respect to say, the Euclidean metric, F does not have finite ε-covers for any $\varepsilon<\frac{1}{4}$. Suppose the contrary, that $C\subseteq F$ is a finite ε-cover for F, for $\varepsilon<\frac{1}{4}$. Let $x_1\leqq x_2\leqq x_3\cdots\leqq x_n$ be the endpoints of the intervals of all the sets h such that $f_h\in C$. Consider

$$g=\left[\frac{1}{2},\frac{\frac{1}{2}+x_1}{2}\right]\cup\left[x_1,\frac{x_1+x_2}{2}\right]\cup\left[x_2,\frac{x_2+x_3}{2}\right]\cdots\cup\left[x_n,\frac{x_n+1}{2}\right].$$

For any $f_h\in C$, notice that f_h and f_g disagree across half of each of the intervals $[x_i,x_{i+1}]$ and the intervals $[\frac{1}{2},x_1]$ and $[x_n,1]$. Thus, f_h and f_g disagree across half the interval $[\frac{1}{2},1]$. It follows that $\int_{x\in\mathbf{R}} L(f_g(x),f_h(x))\,dP\geqq\frac{1}{4}$. Hence, C cannot be a finite ε-cover for F.

Yet, F is easily learnable with respect to P and any metric, as with probability $\frac{1}{2}$, a randomly drawn example for any $f\in F$ will contain a complete encoding of f.

Theorem 6 is the analog of Theorem 2 for the learnability of classes of functions over finite classes of distributions. Notice that Theorem 3 extends the sufficiency conditions of Theorem 2 to infinite classes of distributions, using the notion of entropy for concept classes. Similarly, Theorem 6 can be extended to infinite classes of distributions as follows. Haussler [8] also presents an extension, using the "metric dimension" of a class of functions.

Let P be a distribution on $\mathbf{R}$ and f a function from $\mathbf{R}$ to $\mathbf{R}$. We use P_f to denote the distribution on $\mathbf{R}^2$ given by

$$P_f((x,y))=\begin{cases}P(x) & \text{if } y=f(x),\\ 0 & \text{otherwise.}\end{cases}$$

Analogously for a class of functions F and class of distributions Ξ,

$$\Xi_F=\{P_f\mid P\in\Xi, f\in F\}.$$

Using the above notation, we can state the following theorem.

THEOREM 7. *Let F be a class of functions and L a metric such that $D_L(F)$ is finite. F is solidly learnable over a class of distributions Ξ with respect to L if*

$$\lim_{l\to\infty} \sup_{P\in\Xi_F} \frac{H_{P,l}(\text{graph}(F))}{l} = 0.$$

Proof. Invoking Theorem 3, we have that the concept class graph (F) is solidly learnable over Ξ_F. As in part (1) of Theorem 5, it follows that F is solidly learnable over Ξ with respect to the discrete metric. Finally, as in Corollary 1, it follows that F is solidly learnable over Ξ with respect to any metric for which $D_L(F)$ is finite. □

4. Conclusion. We examined the conditions necessary and sufficient for the learnability of sets and functions over general classes of probability distributions. Our results for the learnability of functions was with respect to general metrics for measuring the distance between two functions. The contributions of this paper as well as related results are presented in Table 1.

TABLE 1

	Finite classes of distributions	Class of all distributions	Infinite classes of distributions
Solid learnability of concept classes		Blumer et al. [4]	This paper
Uniform learnability of concept classes	Benedek and Itai [3]	Blumer et al. [4]	This paper
Solid learnability of function classes		This paper (general metric) Natarajan [12] (discrete metric)	This paper (general metric) Haussler [8] (general metric)
Uniform learnability of function classes	This paper (general metric)	This paper (general metric) Natarajan [12] (discrete metric)	This paper (general metric) Haussler [8] (general metric)

REFERENCES

[1] D. ANGLUIN, *Queries and concept learning*, Machine Learning, 2 (1988), pp. 319-342.

[2] S. BEN-DAVID, G. M. BENEDEK, AND Y. MANSOUR, *A parametrization scheme for classifying models of learnability*, in Proceedings of the 2nd Annual Workshop on Computational Learning Theory, Morgan Kaufmann, San Mateo, CA, 1989, pp. 285-303.

[3] G. BENEDEK AND A. ITAI, *Learnability by fixed distributions*, in Proceedings of the 2nd Annual Workshop on Computational Learning Theory, Morgan Kaufmann, San Mateo, CA, 1988, pp. 80-90.

[4] A. BLUMER, A. EHRENFEUCHT, D. HAUSSLER, AND M. WARMUTH, *Learnability and the Vapnik-Chervonenkis dimension*, J. Assoc. Comput. Mach., 36 (1990), pp. 929-965.

[5] W. FELLER, *An Introduction to Probability Theory and Its Applications*, John Wiley and Sons, New York, 1957.

[6] J. GILL, *Computational complexity of probabilistic Turing machines*, SIAM J. Comput., 6 (1977), pp. 675-695.

[7] D. HAUSSLER, M. KEARNS, N. LITTLESTONE, AND M. WARMUTH, *Equivalence of models for polynomial learnability*, in Proceedings of the 1988 Workshop on Computational Learning Theory, Morgan Kaufmann, San Mateo, CA, 1988, pp. 44–55; Information and Computation, to appear.

[8] D. HAUSSLER, *Generalizing the* PAC *model for neural nets and other learning applications*, Tech. Report UCSC-CRL-89-30, Dept. of Computer Science, University of California, Santa Cruz, CA; also, in Proceedings of the 30th IEEE Symposium on Foundations of Computer Science, pp. 40–55.

[9] M. KEARNS, M. LI, L. PITT, AND L. G. VALIANT, *On the learnability of Boolean formulae*, in Proceedings of the 19th ACM Symposium on Theory of Computing, 1987, ACM Press, NY, pp. 285–295.

[10] B. K. NATARAJAN, *Learning over classes of distributions*, in Proceedings of the Workshop on Computational Learning Theory, 1988, Morgan Kaufmann, San Mateo, CA, pp. 408–409.

[11] ———, *Probably approximate learning over classes of distributions*, Tech. Report HPL-SAL-89-29, Hewlett-Packard Laboratories, Palo Alto, CA, 1989.

[12] ———, *Probably approximate learning of sets and functions*, SIAM J. Comput., 20 (1991), pp. 328–351.

[13] ———, *Machine Learning: A Theoretical Approach*, Morgan Kaufmann, San Mateo, CA, 1991.

[14] B. K. NATARAJAN AND P. TADEPALLI, *Two new frameworks for learning*, in Proceedings of the 5th International Symposium on Machine Learning, Morgan Kaufmann, San Mateo, CA, 1988, pp. 402–415.

[15] L. PITT AND M. WARMUTH, *Reductions among prediction problems: On the difficulty of predicting automata*, in Proceedings of the 3rd Conference on Structure in Complexity Theory, ACM Press, NY, 1988, pp. 60–69.

[16] J. POLLARD, *On the Convergence of Stochastic Processes*, Springer-Verlag, New York, 1986.

[17] R. RIVEST AND R. E. SCHAPIRE, *Diversity-based inference of finite automata*, in Proceedings of the 28th IEEE Symposium on Foundations of Computer Science, IEEE Press, Washington D.C., 1987, pp. 78–87.

[18] L. G. VALIANT, *A theory of the learnable*, Comm. ACM, 27 (1984), pp. 1134–1142.

[19] V. N. VAPNIK, *Inductive principles for empirical dependencies*, in Proceedings of the 2nd Workshop on Computational Learning Theory, Morgan Kaufmann, San Mateo, CA, 1989, pp. 3–24.

[20] V. N. VAPNIK AND A. YA. CHERVONENKIS, *On the uniform convergence of relative frequencies*, Theory of Probability and Its Applications, 16 (1971), pp. 264–280.

[21] ———, *Necessary and sufficient conditions of the uniform convergence of means to their expectations*, Theory Probab. Appl., 26 (1981), pp. 532–553.

SIAM J. COMPUT.
Vol. 21, No. 3, pp. 450-465, June 1992
© 1992 Society for Industrial and Applied Mathematics
003

THE COMPLEXITY OF THE LIN-KERNIGHAN HEURISTIC FOR THE TRAVELING SALESMAN PROBLEM*

CHRISTOS H. PAPADIMITRIOU†

Abstract. It is shown that finding a local optimum solution with respect to the Lin-Kernighan heuristic for the traveling salesman problem is PLS-complete, and thus as hard as any local search problem.

Key words. local search, traveling salesman problem, PLS-complete

1. Introduction. Local search is a powerful approach for obtaining good solutions to hard combinatorial optimization problems (see, for example, [PS] for an exposition). A local search heuristic starts with a solution and repeatedly tries to find a better solution which is a *neighbor* of the first. If a better neighbor is found, a search starts for a better neighbor of that one, and so on. Since problems of this sort have finitely many solutions totally ordered by cost, this process always ends at a *local optimum.* The process may be repeated many times from initial solutions generated in some randomized way. Naturally, the most critical part in the design of such a heuristic is deciding when two solutions are neighbors. A neighborhood should be easy to search, and at the same time rich enough to assure the quality of the local optima obtained.

Perhaps the most famous and successful local search heuristic is the one proposed by Lin and Kernighan in 1973 [LK] for the traveling salesman problem. The Lin-Kernighan heuristic generalizes and optimizes the most obvious neighborhood: 2-change. A tour is a 2-change of another if the two tours differ in two edges (intercity distances traversed, irrespective of sense). The 2-change neighborhood is easy to search, but produces mediocre local optima. 3-change (three edges can be replaced) does quite a bit better. But the real performance breakthrough comes from the main idea in [LK], which is to allow changes of arbitrarily many edges. Among all exponentially many such λ-changes (as such neighbors are called), the Lin-Kernighan heuristic makes an efficient, cost-dependent, breadth-first search for one level and then a depth-first search for a better λ-change. The quality of local optima thus obtained is very impressive. Recent experimental studies [Be] establish the Lin-Kernighan heuristic as the champion among approaches to large traveling salesman problems.

In the past, when the merits of a local search heuristic were debated, it was the quality of local optima that was being disputed, not so much the efficiency of the algorithm, which appeared to be taken for granted. There is good evidence for this. It is reported in [LK] that, in their experience, it takes $O(n^{2.2})$ time to produce a local optimum. Similar reports of empirical efficiency of local search heuristics are common. Nevertheless, it is conceivable that, in the worst case, these algorithms may follow an exponentially long trail of local improvements (such examples are known for the 2-change neighborhood [Lu], and were suspected for all variants). How hard is it to produce a local optimum, for the various neighborhoods?

* Received by the editors January 9, 1991; accepted for publication (in revised form) July 2, 1991. Most of this work was completed while the author was visiting the Computer Technology Institute in Patras, Greece. This research was supported by the ESPRIT Basic Research Action No. 3075 ALCOM, a grant from the Volkswagen Foundation to the Universities of Patras and Bonn, and by the National Science Foundation.

† Department of Computer Science and Engineering, University of California at San Diego, La Jolla, California 92093-0114.

In [JPY] we studied this question in depth and came up with a new complexity class called *PLS*, for *polynomial-time local search*, apparently lying somewhere between P and NP. PLS captures the complexities of local optimization. We showed that certain local optimum problems are PLS-complete, and thus as hard as any such problem (for more recent work on complexity classes closely related with PLS see [MP], [Pa]). The original list of PLS-complete problems in [JPY] was disappointingly short (reflecting difficulties that will become apparent in the current paper as well). It included, besides some artificial, generic problems, just the local search heuristic proposed, also by Kernighan and Lin, for the graph partitioning problem [KL]. Most conspicuously absent was the Lin–Kernighan heuristic for the traveling salesman problem, the local search heuristic *par excellence*, and in many ways the motivating example for the work reported in [JPY]. Quoting from [JPY]:

> What we would most like to prove is that the Lin–Kernighan λ-change algorithm for the traveling salesman problem is PLS-complete.... Unfortunately, we at present see no way of extending our techniques to this problem.

In the present paper we prove this conjecture. We define a local search algorithm for the traveling salesman problem that captures the spirit of the Lin–Kernighan heuristic, and show that finding a local optimum under this algorithm is PLS-complete. For the proof we build on some very technical work by Krentel, Schäffer, and Yannakakis, that has been going on since the appearance of [JPY]. First, Krentel [Kr1] showed that it is PLS-complete to find a local optimum for the problem of maximum satisfiability with weights on the clauses (where two truth assignments are considered neighbors if they agree on all variables but one). Then, Schäffer and Yannakakis [SY] showed that the result is true even when all clauses have just two literals (this has the important implication that obtaining a stable configuration in Hopfield neural nets [Ho] is PLS-complete). Our reduction starts from the two-literal result.

2. Definitions. A problem Π in PLS is characterized by several parameters. First, for each input $x \in \Sigma^*$ (e.g., a distance matrix for the traveling salesman problem) we have a set of feasible solutions F_x (e.g., tours in the traveling salesman problem) such that, given x and s, it is easy to decide whether $s \in F_x$. Then, given x, we can in polynomial time produce a feasible solution $s_0 \in F_x$ (e.g., for the traveling salesman problem, the tour visiting the cities in numerical order). Next, given x and $s \in F_x$, we can compute in polynomial time the *cost* of s (e.g., the length of the tour). Finally, given an x and an $s \in F_x$, we can test in polynomial time whether s is local optimum, and, if not, produce a solution with better cost (for the example of the traveling salesman problem, the tour generated by the Lin–Kernighan heuristic formalized below). Π is the following computational problem: *Given an input x, find a locally optimal solution $s \in F_x$.*

2SATFLIP is an important problem in PLS. We are given a set of clauses $C_1, \cdots, C_m$, each involving two literals (variables or negations) from $x_1, \cdots, x_n$. Each clause C_i has an integer weight $W_i > 0$ assigned to it. A solution is, naturally, a truth assignment to the n variables. The cost of a solution is the sum of the weights of the clauses satisfied by the assignment. Finally, an assignment is locally optimal if, by flipping any of the n variables, no improvement in the cost can be made. 2SATFLIP is known to be PLS-complete [SY].

In this paper we show that the problem LIN–KERNIGHAN defined below is PLS-complete by giving a *PLS-reduction* from 2SATFLIP to it. A PLS-reduction from problem A in PLS to another problem B in PLS is defined in terms of two polynomially

computable functions f and g. Given an instance x of A, f computes an instance $f(x)$ of B such that, for any local optimum s of $f(x)$, $g(s)$ is a local optimum of x.

The problem LIN-KERNIGHAN has instances, feasible solutions, and costs as usual in the traveling salesman problem. The innovation in the Lin-Kernighan heuristic is their definition of the neighborhood. There are a number of difficulties. First, the neighborhood structure of LIN-KERNIGHAN is complex, asymmetric, and cost-dependent. More seriously, this neighborhood is defined via a very complicated algorithm, with unspecified parameters and details; there are several existing variants and implementations. In the sequel we define a stylized version of the Lin-Kernighan heuristic, which, we believe, captures the spirit of the approach, while being relatively clean. We shall next describe how "our" Lin-Kernighan heuristic searches for an improved neighbor of a given tour. (Our goal in the description of the algorithm is clarity, and not efficiency and programming style or succinctness.) The tour is considered as a set T of edges. The cost of edge e is denoted by $c(e)$, and the cost of a set of edges S by $c(S)$. The search of a better neighbor of T proceeds as follows:

Lin–Kernighan heuristic for improving tour T:

Step 1. All 2-changes and 3-changes (tours $T' = (T - X) \cup Y$ with $|X| = |Y| \leqq 3$) of T are tried. If one is found with $c(T') < c(T)$, it is the answer sought.

Step 2. We repeat Step 3 for all combinations of edges $x_1 \in T$ and $y_1 \notin T$ such that (a) the two edges are adjacent, and (b) $c(y_1) < c(x_1)$ (this is the top-level, breadth-first part of the search; the rest (Step 3) is depth-first).

Comment. Notice that, once x_1 is removed, T becomes a Hamilton path. Step 3 deals with such a Hamilton path. One of its endpoints (currently the common node of x_1 and y_1) is the *active* endpoint, and will change as the algorithm proceeds; the other will be inactive throughout. Once y_i (in our case, $i = 1$) has been added to the Hamilton path, the next edge to go so that a Hamilton path results, x_{i+1}, is determined uniquely (see Fig. 1). Also, once x_{i+1} is determined, the active endpoint changes, and there is a unique edge $\hat{y}_{i+1}$ that completes the tour (joins the two endpoints).

Recall that, so far, we have determined edges x_1 and x_2 to be deleted from T, and edge y_1 to be added. $i := 1$.

Step 3. Set $i := i + 1$. Consider all edges $y_i \notin T$ leaving the active endpoint such that $\sum_{j=1}^{i} [c(y_i) - c(x_i)] > 0$ (we call this the *positive sum criterion*). If none, then go to Step 4. Among all those edges, choose the y_i that maximizes the difference $c(y_i) - c(x_{i+1})$ (recall that y_i uniquely determines x_{i+1}; we call this the *next best criterion*). Break ties lexicographically. Repeat Step 3.

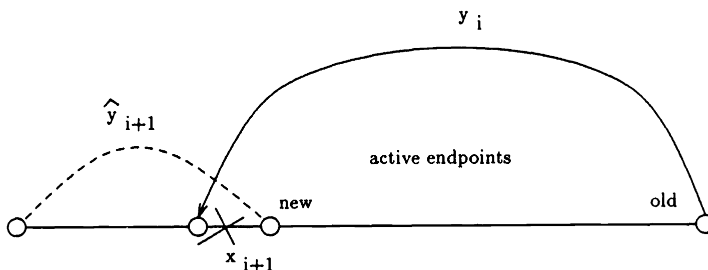


FIG. 1. *The Lin-Kernighan heuristic.*

Step 4. For $j := 2, \cdots, i$, let $T_j = T - \{x_1, \cdots, x_j\} \cup \{y_1, \cdots, \hat{y}_j\}$. That is, T_j is the tour that would result if we stopped the procedure at $i = j$ and joined the endpoints. Choose the T_j with the smallest cost. If $c(T_j) < c(T)$, then T_j is the better neighbor sought. Otherwise continue (with the next choice of x_1 and y_1, Step 2).

Step 5. If all choices for x_1 and y_1 were tried and no better neighbor was found, T is a local optimum.

This completes our definition of problem LIN-KERNIGHAN: Given an instance of the traveling salesman problem, produce a tour such that the above algorithm fails to improve it. It should be clear that we can search for a Lin–Kernighan improvement in polynomial time.

The Lin–Kernighan algorithm as defined above differs in many details from the one described in [LK]. Perhaps the most notable departure of the algorithm above from the implementation described in [LK] is that we allow edges that enter the tour to depart again, but disallow edges that leave the tour to re-enter; in [LK] we have exactly the opposite. Our construction depends on this feature (compare the two changes in the left of Fig. 7). We feel that the PLS-completeness of the problem is a consequence of the complexity of the traveling salesman problem, and the Lin–Kernighan philosophy of telescoping changes that are disadvantageous in the short term; thus, we conjecture that the result is true of any other reasonable variant or implementation (although by this we do not mean that we know precisely how to modify the proof for each).

3. The construction. We now prove our main result.

THEOREM. LIN-KERNIGHAN *is* PLS-*complete.*

We shall PLS-reduce 2SATFLIP to LIN-KERNIGHAN. We have to describe two polynomially computable functions f and g such that, given an input x to 2SATFLIP (a set of 2-clauses and weights), $f(x)$ is a traveling salesman problem instance, such that s is a Lin–Kernighan local optimum of $f(x)$ if and only if $g(s)$ is a local optimum of x. We shall describe f, that is, we shall show how to construct an instance of the traveling salesman problem starting from any weighted set of 2-clauses. The function g, recovering the local optimum, will follow from this construction and Lemmata 1 and 2 below.

General description. We first give a high-level description of the traveling salesman problem constructed. All edges have huge costs, except for the edges of a sparse graph, which have reasonable costs (the precise costs will be specified later as needed, see Fig. 8). We shall describe the graph of reasonably priced edges first (see Fig. 2; all other "edges" will be referred to henceforth as *nonedges*).

For each variable we have two parallel paths, corresponding to the values true and false (left side of Fig. 2). A tour traversing one of these paths can be thought of as assigning a truth value to the variables. For each clause C_i we have an edge $a_i - b_i$ on the right. All these edges and paths are connected in series, via paths of length three, and the structure is closed by the diamond on the upper right in Fig. 2. The lower node of the diamond is connected by edges with the upper common nodes of all pairs of parallel paths corresponding to variables; these edges are called "start" edges. The side edges of the diamond, called "bait" edges, are the most expensive among the reasonable edges being described.

Naturally, this is not all: The structure of the clauses must be reflected in the construction. For each clause C_i consisting of two literals, we have an OR subgraph (Fig. 3) connecting the paths corresponding to the two literals. It is easy to see that

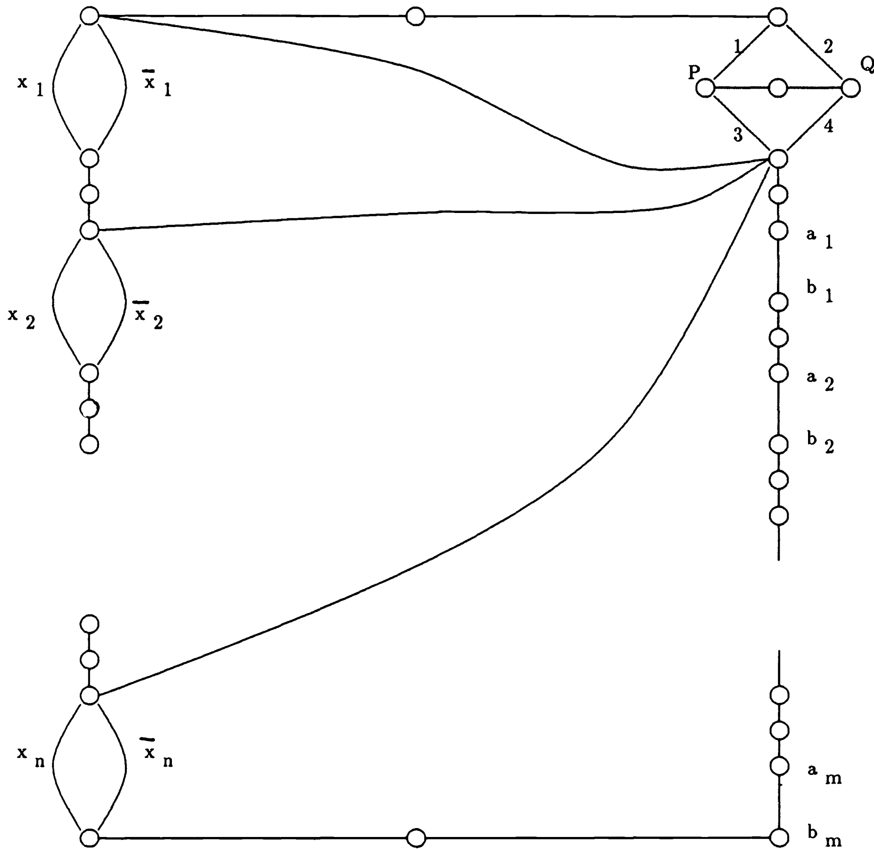


FIG. 2. *The general structure of the* TSP.

an OR subgraph (connected to the rest of the graph only by its four endpoints) can be traversed by a tour in the three ways shown, corresponding to *upper literal true* Fig. 3(b), *lower literal true* Fig. 3(c), and *both literals true* Fig. 3(d). Thus the path corresponding to a literal is in fact a series of such devices, connected as in Fig. 4. The additional edges connecting the first nodes of consecutive OR devices and the last node of the path are called *jump edges*, and are instrumental to guiding the Lin-Kernighan heuristic. They have weight -1 (Fig. 8).

Since each OR device connects the paths standing for the truth values of the literals of a clause, if these were the only possible traversals of the device, the clause would be assigned the value true. However, there is yet another way to traverse this device, say, corresponding to clause C_i: It can be "picked up" by a path from a_i to b_i (Fig. 5; recall, $a_i - b_i$ is the edge on the rightside of Fig. 2 corresponding to clause C_i). In fact, it can be picked up in two ways. However, the edges leaving a_i and b_i to pick up the clause (called "penalty edges") are more expensive than the others, *proportionally to the weight W_i of clause C_i* (see Fig. 8). Thus, if the OR device corresponding to C_i is picked up this way, that is, if C_i is not satisfied by the assignment suggested by the tour, then an extra cost proportional to W_i is paid, exactly as appropriate.

Finally, let us discuss the function of the diamond. Obviously, there are two ways that a tour can traverse the diamond. Suppose that T traverses edges 1 and 4 (Fig. 2). Then a possible λ-change of T may start by deleting edge 4 of the diamond (a "bait"

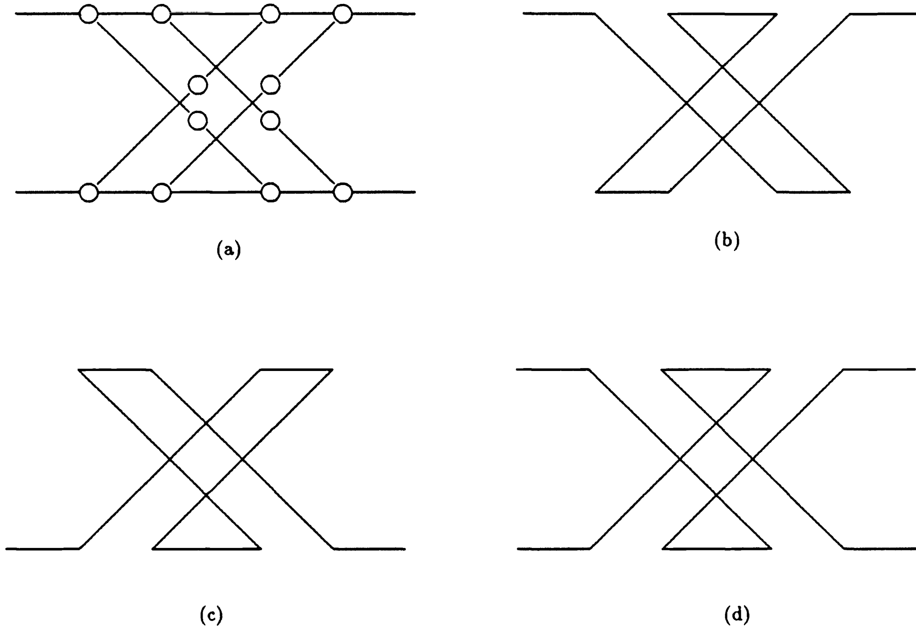


FIG. 3. *The* OR *device.*

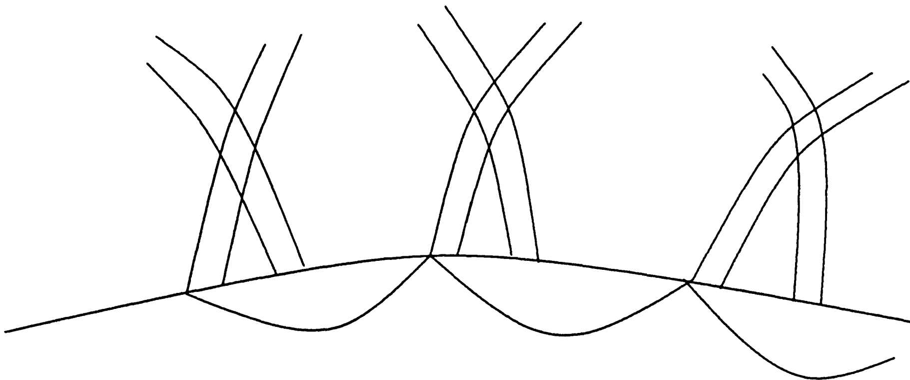


FIG 4. *Literal path with jump edges.*

edge) and adding to the tour one of the "start" edges, say, the one leading to the upper node of variable x_j. This can be thought of as *flipping* the truth value of x_j. We shall see that, after this, the Lin-Kernighan heuristic actually implements this flipping at the variable and clause parts. After this is done, the new tour is closed by traversing the diamond in the other way (edges 2 and 3; that is, the traversal of the diamond flips from one λ-change to the next). The huge gains obtained from the deletion of bait edge 4, which encouraged exploration of the flipping of the truth value of x_i, were annulled by the addition of bait edge 3. The λ-change is favorable only if the new truth assignment is better than the old one (that is, the weight of the satisfied clauses increases).

Let us review once more the high-level description of the construction: All edges except for those shown in Figs. 2, 3, and 4 have huge costs. The graph in Fig. 2 can be traversed by a tour as follows (see Fig. 6): The diamond is traversed in one of the two ways. For each variable, either the true path or the false path is taken. For the

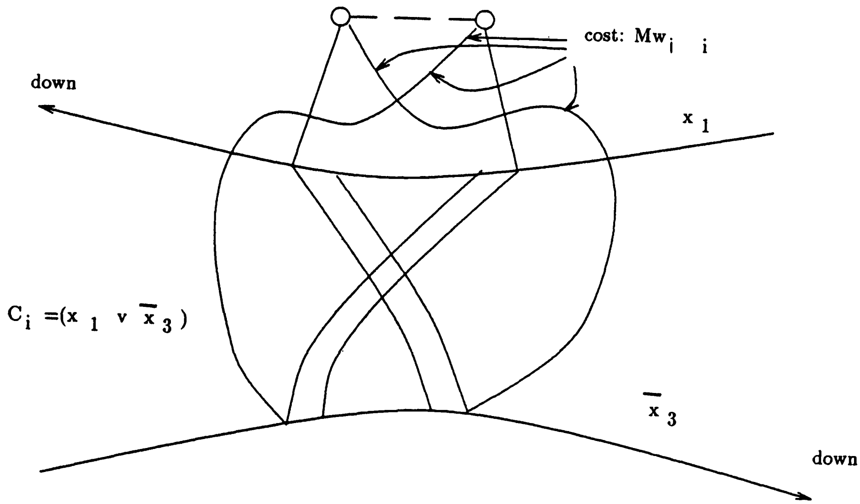


FIG. 5. *Picking up an unsatisfied clause.*

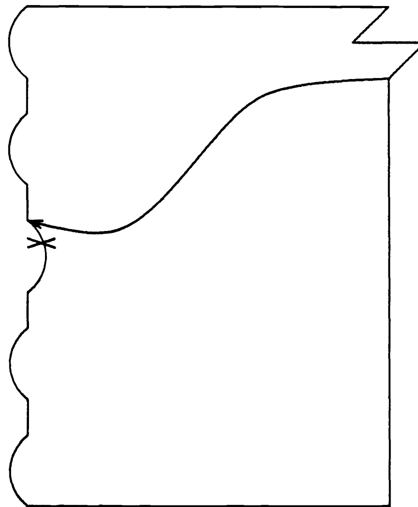


FIG. 6. *A standard tour.*

OR device corresponding to a clause there are four possibilities: Either it is picked up by the first literal, or by the second, or by both (Fig. 3), or finally it may be picked up (at a cost proportional to the weight of the clause) by an $a_i - b_i$ path (Fig. 5). The tour is closed through the right side, where the edge $a_i - b_i$ is followed for all satisfied clauses; otherwise the path $a_i - b_i$ is followed, as above. Such a tour is called a *standard tour*, and it corresponds in a natural way with a truth assignment of the original instance of 2SATFLIP. *This natural correspondence will be our function g.* Also, notice that the cost of a standard tour is a constant depending only on m and n, plus $M \sum_j W_j$, where the sum is taken over all clauses satisfied by the corresponding truth assignment.

We need to prove two things.

LEMMA 1. *A standard tour T is a local optimum only if g(T) (that is, the corresponding truth assignment) is a local optimum.*

LEMMA 2. *The only local optimum tours are the standard tours.*

It is easy to see that the main theorem follows from these two lemmas.

Simulating a flip. In this section we prove Lemma 1. We prove that, if $g(T)$ is not a local optimum, that is, whenever there is a variable x_i whose flipping improves the weight of the satisfied clauses, then the standard tour T is not locally optimal under Lin-Kernighan either. We do this by showing how the Lin-Kernighan heuristic would improve T. In other words, *we simulate a flip by a Lin-Kernighan improvement.*

Suppose, without loss of generality, that the proposed flipping of x_i is from true to false (that is, x_i is true in $g(T)$). Our Lin-Kernighan improvement (recall the algorithm in § 2) starts by deleting the bait edge in T, and adding the starting edge leading to x_i (see Fig. 6). Since the bait edge is more expensive than the start edge (see Fig. 8), it will be one of the possibilities tried (recall Step 2 of the algorithm). This immediately determines the next edge to be deleted: It is the first edge of the path corresponding to the truth value $x_i = \text{true}$. The simulation of a flip by the Lin-Kernighan heuristic has started. The currently active node is the leftmost node of an OR, and the next best edge to be added is the jump edge (Figs. 7 and 9).

There are two possibilities: The clause corresponding to the OR is either satisfied by both x_i and the other literal, or it is satisfied by x_i alone. In the first case, the Lin-Kernighan search follows unambiguously the changes in Fig. 7, and the end result is that the clause is now satisfied by the other literal alone. In the second case, the search follows the changes shown in Fig. 9, resulting in an unsatisfied clause (and the additional cost $2MW_j$). Notice that the jump edges were added and then deleted; this is allowed in Lin-Kernighan, but the converse is not (recall the condition $y_i \notin T$ in

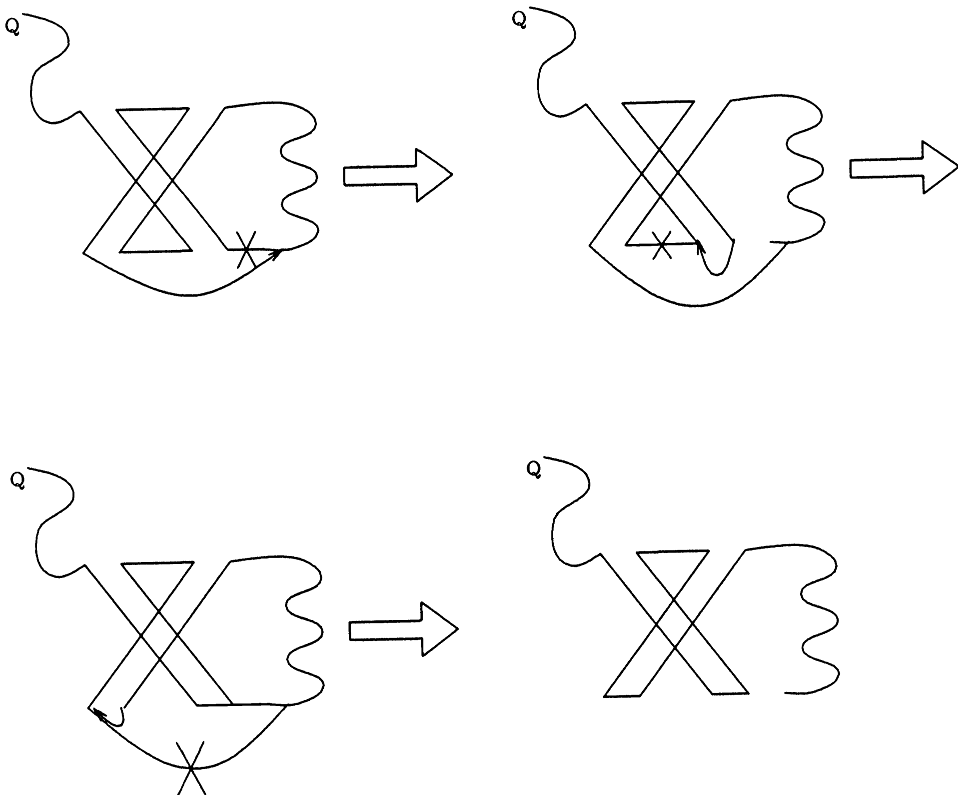


FIG. 7. *Clause twice satisfied.*

Forced edges	$-M^4$
Bait edges	M^3
Start edges	M^2
Penalty edges (clause C_j)	MW_j
Jump edges	-1
All other edges	0
Nonedges	
$v_2 - *$	M^9
$P - *, Q - *$	M^8
$v_1 - v_1', v_3 - v_3'$	M^7
OR-*	M^6
a_i / b_i-*	M^5
Other $v_1 - v_3'$ nonedges	M^4

FIG. 8. *The costs.*

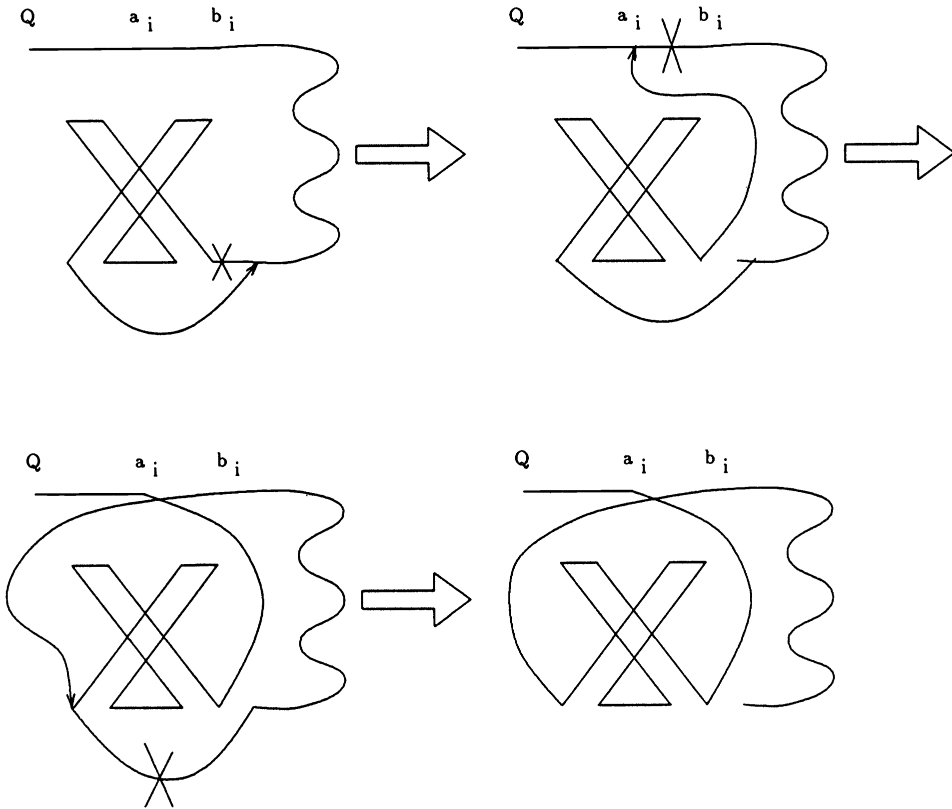


FIG. 9. *Clause C_j satisfied by x_i alone.*

Step 3 of the Algorithm). In the end of either case, the active endpoint is at the same position at the next OR of the path of x_i, and the process (changes in Fig. 7 or Fig. 9) will be repeated. It is easy to check (but also necessary for our argument) that the next best criterion and the lengths of the edges force the course described. Finally, the end of the x_i path will be reached, and the lower end of the path will become active (Fig. 10). There is no choice but to add the last jump edge of the path corresponding to $\bar{x}_i$. This way, we start on the inverse process, that of satisfying the clauses involving $\bar{x}_i$.

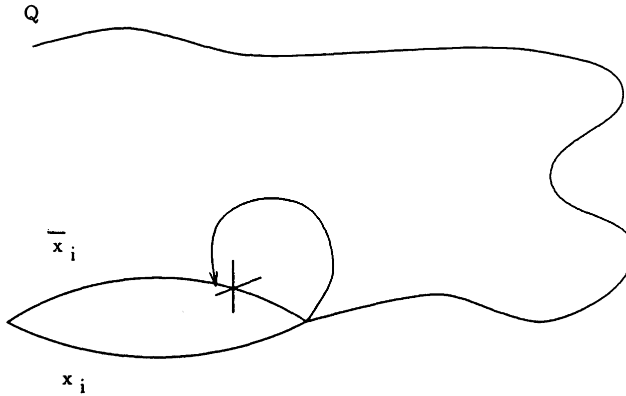


FIG. 10. *Starting the $\bar{x}_i$ path.*

Suppose that the next clause involving $\bar{x}_i$ is already satisfied by the other literal in it. Then the Lin–Kernighan search will follow the changes in Fig. 11. If it is not satisfied, and it is picked up by penalty edges, then Fig. 12 is followed. If, however, the OR is picked up by penalty edges from *the opposite side of* $\bar{x}_i$ (recall the two ways of picking up an OR in Fig. 5), then Fig. 13 is followed. The final result is that, if a previously unsatisfied clause C_j contained x_i, it is now satisfied, and $2MW_j$ is saved.

Proceeding this way we flip the truth value of x_i, and end up back at the start edge. The only feasible continuation is shown in Fig. 14. The start edge is deleted, a

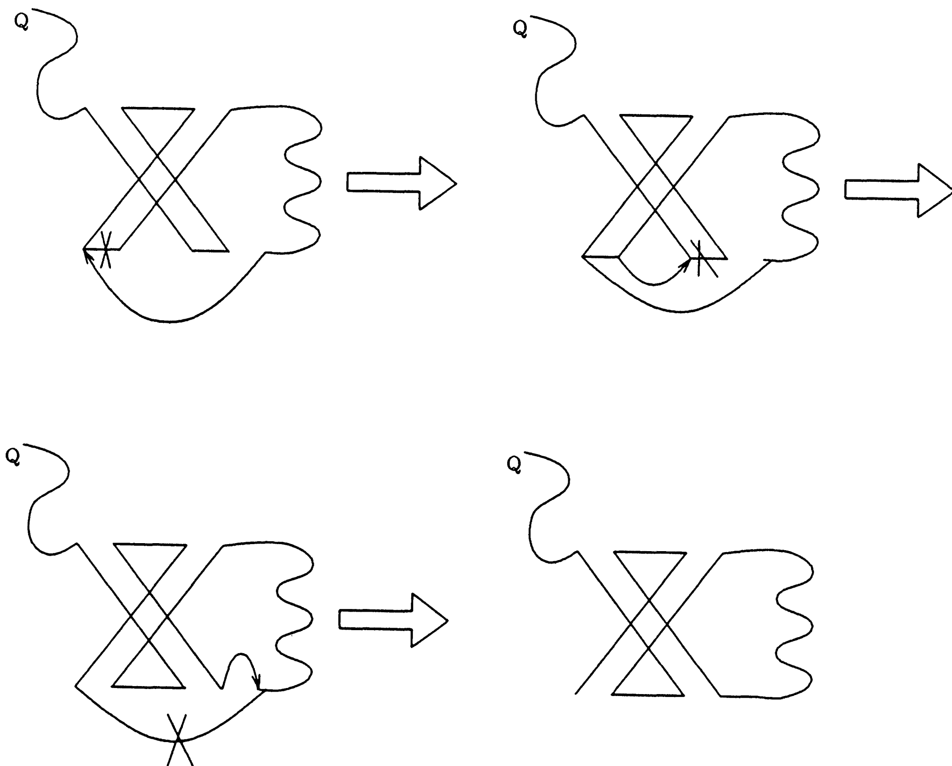


FIG. 11. *Clause already satisfied.*

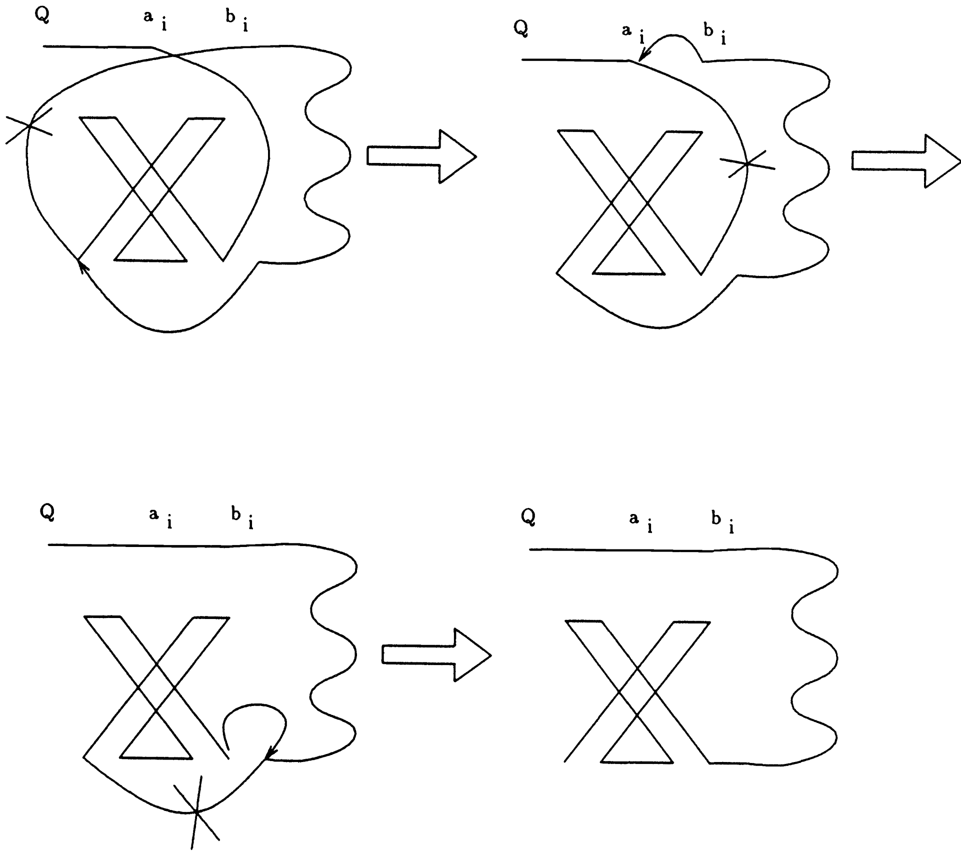


FIG. 12. *Clause unsatisfied.*

new bait edge is added, and the diamond is traversed in a new way. Since this is no longer the breadth-first stage (Step 2), there is no incentive to try another start edge and flip another variable. The only potentially satisfactory completion of the change is the one indicated in Fig. 14. It is an improvement of the starting tour if and only if flipping x_i generated a profit. The proof of Lemma 1 is complete. □

Excluding other tours. In this subsection we prove Lemma 2. We shall show that any tour that is not standard can be improved by Lin–Kernighan. Let us start with a very basic observation: The graph in Figs. 2, 3, and 4 (the graph of "reasonably priced edges") is in some sense "tripartite." Exactly one-third of its nodes have degree 2 (see Figs. 2 and 3). They are adjacent each to two other nodes of degree at least three. Thus, we can say that the graph consists of several paths of the form $v_1 - v_2 - v_3$, with their endpoints v_1 and v_3 connected in some way. It is also important to notice that each of these paths is traversed in a particular direction in any standard tour (except, of course, for the $P - Q$ path, for which the two directions flip-flop), and hence it is unambiguous which node is the v_1 node and which the v_3 node. We shall call the edges $v_1 - v_2$ and $v_2 - v_3$ the "forced" edges of the graph. Their cost is extremely attractive ($-M^4$, see Fig. 8).

Because of the huge costs of nonedges out of the v_2 nodes, P, and Q (Fig. 8), in any locally optimum tour all forced edges will be present (if one is missing, then it is easy to see that a 2-change will correct that), and the diamond will be traversed in

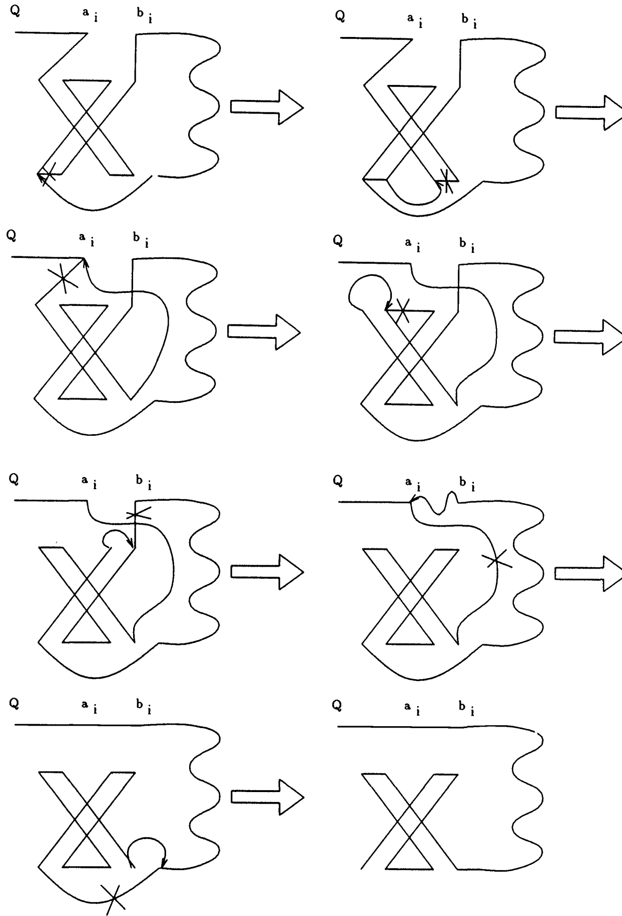


FIG. 13. *Clause unsatisfied, picked up the wrong way.*

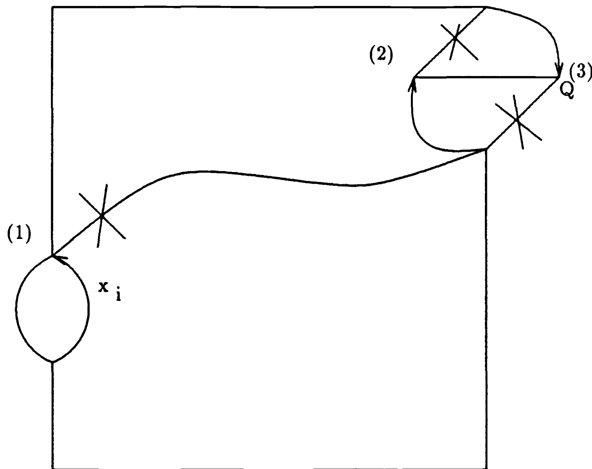


FIG. 14. *Completing the change.*

one of the two intended ways. Thus there are no nonedges of cost higher than M^7 in a locally optimal tour.

Suppose that there is an M^7 nonedge of the form $v_1 - v_1'$ in the tour. Then the 3-change in Fig. 15 (2-change if v_3' is followed directly in the tour by a v_3'' node) removes it, and symmetrically for any $v_3 - v_3$ edge. Hence the only possible edges connecting two v_3 or two v_1 nodes in a tour are jump edges. Next, notice that all jump edges are of the form $v_1 - v_1'$. Since in a tour there must be an equal number of $v_1 - v_1'$ and $v_3 - v_3'$ edges, there are no such edges whatever.

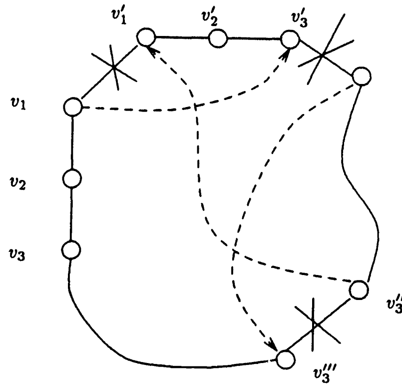


FIG. 15. *Removing $v_1 - v_1'$ edges.*

Recall the OR graph in Fig. 3. It has twelve nodes: The four degree-2 nodes, the four outer nodes (left and right, up and down), and the four *OR nodes* (middle, up, and down). The next heaviest nonedges are the $v_1 - v_3'$ nonedges connecting an OR node of an OR device to another node (called OR-* edges). We shall next show that they cannot be part of any locally optimal tour. The reasoning is as follows: Consider an OR-* edge on the tour (Fig. 16). Next to it we have a 0-cost edge, not participating in the tour, that leads to another OR node. If the nonforced edge out of the other endpoint of the 0 edge is as in Fig. 16(a), then a 2-change removes the OR-* edge. Otherwise (Fig. 16(b)) a 3-change does. Hence there are no OR-* edges. This implies that all OR graphs are traversed in one of the intended ways shown in Fig. 3.

We next make sure that the a_i's and b_i's are traversed as intended. All nonedges out of such a node cost a huge M^5. Either one or two such edges leave an $a_i - b_i$ pair. In both cases an improvement by a 2- or 3-change is possible, exactly as in Fig. 16.

We have therefore established that in any locally optimal tour the forced edges, the diamond, and all OR devices are traversed as intended. Also, the $a_i - b_i$ pairs are traversed by edges: either by the $a_i - b_i$ edge, or by one of the two pairs of penalty edges. It follows that such a tour differs from a standard one in that it does not visit whole paths of the variable side, but *fragments* of paths (see Fig. 17 for a schematic description). Each path fragment ends with an M^4 nonedge. It is easy to see, in view of the proof of Lemma 1, that such a nonstandard tour can be made standard by Lin–Kernighan. The improvement starts as shown in Fig. 18, and continues as in Figs. 7–13.

This concludes the proof of Lemma 2, and the theorem. □

There are two interesting consequences of the theorem that follow directly from the techniques of [SY] and [PSY]: First, one can construct examples of traveling

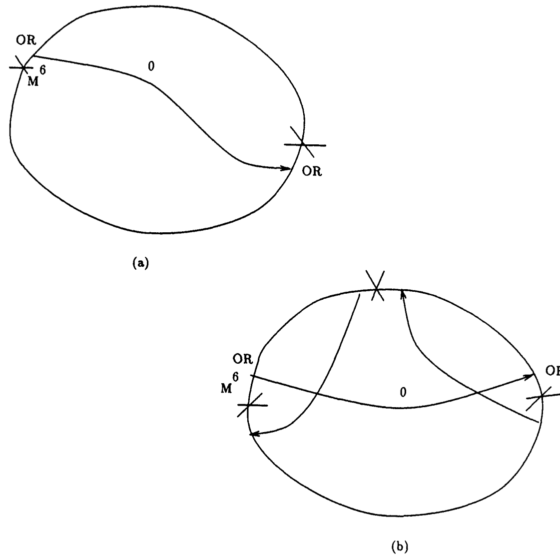


FIG. 16. *Removing* OR-* *edges.*

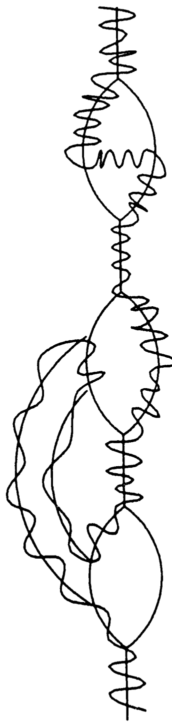
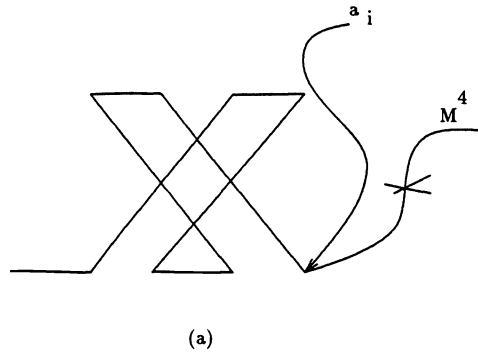
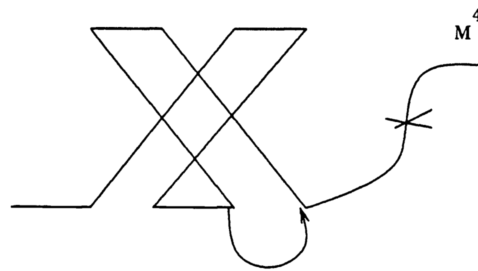


FIG. 17. *Nonstandard traversal of variables.*



(a)



(b)

FIG. 18. *Start of the improvement.*

saleman problems such that the Lin–Kernighan heuristic (actually, the variant studied here) takes exponentially long to converge. The existence of such instances had been open. Second, it is a PSPACE-complete problem, given a starting tour, to find any local optimum reachable from this tour. Therefore, the problem of obtaining local optima from given starting points according to the Lin–Kernighan neighborhood is perhaps the PSPACE-complete problem that has been solved with the most empirical success.

As an open problem, it would be interesting to show the same result for a variant of Lin–Kernighan in which no added edge is deleted, and no deleted edge is added. This would present several difficulties in the design of the devices, but we conjecture that such a construction is possible. Also, recently Krentel showed a related result, namely, that the k-change neighborhood for the traveling salesman problem, for some huge value of k, also gives rise to a PLS-complete problem [Kr2]. Although such a neighborhood is not useful in practice, the result is still interesting, at least as a promise for a similar result with $k=3$ or 2 (these are the most natural neighborhoods for the traveling salesman problem, still of practical importance). Krentel's reduction starts from clauses *with a bounded number of appearances of each variable.* It seems that a version of our construction would yield a similar result, with a much better constant k. However, this constant will be still in the dozens, and thus the result does not correspond to a practically important heuristic. It would be interesting to show that finding local optima under 2- and 3-changes in the traveling salesman problem are PLS-complete problems as well.

Acknowledgments. Many thanks are due to David Johnson, Mark Krentel, Alex Schäffer, and/or to several anonymous referees, for their helpful criticism of this paper.

REFERENCES

[Be] J. L. BENTLEY, Invited Talk at the First SIAM Symposium on Discrete Algorithms, 1990.

[GJ] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman, San Francisco, CA, 1979.

[Ho] J. J. HOPFIELD, *Neural networks and physical systems with emergent collective computational capabilities*, Proc. Nat. Acad. Sci., 79 (1982), pp. 2554-2558.

[JPY] D. S. JOHNSON, C. H. PAPADIMITRIOU, AND M. YANNAKAKIS, *How easy is local search?*, in Proceedings of Foundations of Computer Science, 1985; also in J. CSS. 37 (1988), pp. 79-100.

[KL] B. W. KERNIGHAN AND S. LIN, *An efficient heuristic procedure for partitioning graphs*, Bell S. T. J., 49 (1970), pp. 291-307.

[Kr1] M. W. KRENTEL, *On Finding Locally Optimal Solutions*, in Proceedings of the 4th Structures Conference, 1989.

[Kr2] ———, manuscript, 1989.

[LK] S. LIN AND B. W. KERNIGHAN, *An effective heuristic for the traveling salesman problem*, Operations Res., 21 (1973), pp. 489-516.

[Lu] G. LUEKER, manuscript, Princeton University, Princeton, NJ, 1975.

[MP] N. MEGIDDO AND C. H. PAPADIMITRIOU, *On total functions, existence theorems, and computational complexity*, Theoret. Comput. Sci., 81 (1981), pp. 317-324.

[Pa] C. H. PAPADIMITRIOU, *On graph-theoretic lemmata and complexity classes*, in Proceedings on Foundations of Computer Science, 1990.

[PS] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[PSY] C. H. PAPADIMITRIOU, A. A. SCHÄFFER, AND M. YANNAKAKIS, *On the complexity of local search*, Proc. 1990 STOC Conference.

[SY] A. A. SCHÄFFER AND M. YANNAKAKIS, *Simple local search problems that are hard to solve*, SIAM J. Comput., 20 (1990), pp. 56-87.

SIAM J. COMPUT.
Vol. 21, No. 3, pp. 466–485, June 1992

© 1992 Society for Industrial and Applied Mathematics
004

OPTIMAL PARALLEL RANDOMIZED ALGORITHMS FOR THREE-DIMENSIONAL CONVEX HULLS AND RELATED PROBLEMS*

JOHN H. REIF† AND SANDEEP SEN†

Abstract. Further applications of random sampling techniques which have been used for deriving efficient parallel algorithms are presented by J. H. Reif and S. Sen [*Proc. 16th International Conference on Parallel Processing*, 1987]. This paper presents an optimal parallel randomized algorithm for computing intersection of half spaces in three dimensions. Because of well-known reductions, these methods also yield equally efficient algorithms for fundamental problems like the convex hull in three dimensions, Voronoi diagram of point sites on a plane, and Euclidean minimal spanning tree. The algorithms run in time $T = O(\log n)$ for worst-case inputs and use $P = O(n)$ processors in a CREW PRAM model where n is the input size. They are randomized in the sense that they use a total of only polylogarithmic number of random bits and terminate in the claimed time bound with probability $1 - n^{-\alpha}$ for any fixed $\alpha > 0$. They are also optimal in $P \cdot T$ product since the sequential time bound for all these problems is $\Omega(n \log n)$. The best known deterministic parallel algorithms for two-dimensional Voronoi-diagram and three-dimensional convex hull run in $O(\log^2 n)$ and $O(\log^2 n \log^* n)$ time, respectively, while using $O(n/\log n)$ and $O(n)$ processors, respectively.

Key words. convex-hulls, parallel algorithms, randomization, computational geometry

AMS(MOS) subject classifications. 68E05, 68C05, 68C25

1. Introduction.

1.1. Background and previous work. Designing efficient parallel algorithms for various fundamental problems in computational geometry has received much attention in the last few years. After some early work by Chow [6] in her thesis, Aggarwal et al. [1] developed some general techniques for designing efficient parallel algorithms for fundamental geometric problems. Most of the problems tackled in that paper had $\Theta(n \log n)$ sequential complexity and the authors presented parallel algorithms which used a linear number of processors and ran in $O(\log^k n)$ time (k being typically 2, 3, or 4) in size of the input. Consequently, a majority of the algorithms were not optimal in $P \cdot T$ bounds. A number of the problems in the original list (in [1]) have now been successfully resolved as far as $O(\log n)$ time, n processors algorithms are concerned, mainly due to work by Atallah, Cole, and Goodrich [2]. They extended the techniques used by Cole [12] for his parallel mergesort algorithm and used a date-structure called *plane-sweep tree* (first proposed by Aggarwal et al. [1]) to arrive at the optimal algorithms. Perhaps the two most important problems which have been eluding such efforts are the two-dimensional Voronoi diagram problem and the convex hull of points in 3-space. These are very fundamental problems in computational geometry and optimal algorithms for these problems would imply corresponding optimal solutions for a multitude of other problems.

A very general definition of Voronoi diagram given by Edelsbrunner [19] is as follows:

> Let S be a finite set of subsets of E^d and for each $s \in$ S let d_s be a mapping of E^d to positive real numbers; we call $d_s(p)$ the distance function of s. The set $\{p \in E^d$: $d_s(p) < d_t(p), t \in S$ - {s}$\}$ is the Voronoi cell of s and the cell complex defined by the *Voronoi cells* of all subsets in S is called the *Voronoi diagram* of S.

*Received by the editors March 29, 1990; accepted for publication (in revised form) May 14, 1991. This research was supported in part by Air Force contract AFSOR-87-0386, Office of Naval Research contract N00014-87-K-0310, National Science Foundation contract CCR-8696134, Defense Advanced Research Projects Agency/Army Research Office contract DAAL03-88-K-0185, DARPA/ISTO contract N00014-88-K-0458. A preliminary version of this paper appeared as *Polling: A new randomized sampling technique for computational geometry* in the 21st Annual Symposium on Theory of Computing, Seattle, WA, 1989.

†Computer Science Department, Duke University, Durham, North Carolina 27706.

In this paper, we confine ourselves to the case where S is a set of points in E^2 and the distance function is the L_2 metric. In mathematical literature, Voronoi diagrams appeared as early as 1850 (due to Dirichlet) and again in 1907 due to Voronoi. Problems about packing and covering of space by balls and other convex figures were among the first major applications of such diagrams. Shamos and Hoey [32] introduced Voronoi diagrams to computer science and and since then a considerable amount of research has been devoted to deriving efficient sequential algorithms for the two-dimensional Voronoi diagram problem [20], [30], [10]. The Voronoi diagram is a very versatile tool for obtaining efficient solutions of some important proximity problems and is also a fundamental mathematical object in its own right. A large number of the problems can be solved in linear or $O(n \log n)$ time from the information contained in the Voronoi diagram that includes *all-points nearest neighbor, Euclidean minimal spanning tree, diameter, smallest enclosing circle*, among others. In §6.1, we make use of this property to obtain efficient parallel algorithms for some of these problems.

Since there are sequential algorithms for Voronoi diagrams that run in time $\Theta(n \log n)$, it is a fundamental question whether there is a parallel algorithm that runs in $O(\log n)$ time using n processors. Aggarwal et al. have given an $O(\log^2 n)$ time, $O(n)$ processors algorithm which was recently improved to $n/\log n$ processors by Goodrich et al. [26] (or alternately $O(\log n \log\log n)$ time and $O(n \log^2 n)$ work). However, it appears that very different techniques would be required to eliminate the $O(\log n)$ factor. Cole and Goodrich [13] reiterated the difficulties posed by this problem when they provided some more applications of their cascaded-merging technique but were unable to extend it to the Voronoi diagram problem. In this paper we settle this question by presenting a randomized algorithm for this problem that runs in $O(\log n)$ time and uses n processors in a shared memory model of parallel computation. The reader should note that the lower bound of $\Omega(n \log n)$ also applies to the randomized algorithms by a reduction of sorting to (one-dimensional) Voronoi diagrams. Levcopoulos, Katajainen, and Lingas [24] presented an optimal expected time algorithm for Voronoi diagrams for a randomly chosen set of input points; in contrast our algorithm makes no assumption about the input distribution and is optimal for the worst-case input.

Convex hulls in three-dimensions has a wide range of applications ranging from computer graphics to design automation, pattern recognition, and operations research. Convex hulls in three dimensions can also be constructed sequentially in $\Theta(n \log n)$ time where as the best known deterministic parallel algorithm due to Dadoun and Kirkpatrick [14] runs in $O(\log^2 n \log^* n)$ time using n processors. In this paper we actually describe an optimal randomized parallel algorithm for constructing convex hulls in Euclidean 3-space. Due to a well-known reduction from two-dimensional Voronoi diagrams to three-dimensional convex hulls, we get an equally efficient algorithm for the first problem as an immediate corollary.

1.2. Random-sampling and polling in computational geometry. Randomization has been successfully used in a wide number of applications (for example, see [21], [29], and [33]) and has recently been used to obtain efficient algorithms in computational geometry. Clarkson [8]–[10], Haussler and Welzl [22], and Mulmuley [25] used random sampling techniques to derive better upper-bounds for a large number of problems including the post office problem, higher-order Voronoi diagrams, segment intersections, linear programming, and higher-dimensional convex hulls. The general approach taken by these algorithms is as follows: a randomly chosen subset R of the input set S is used to partition the problem into smaller ones. Clarkson [10] proved that for a wide class of problems in computational geometry, the *expected* size of each subproblem is $O(|S|/|R|)$

and moreover the *expected* total size of the subproblems is $O(|S|)$. A random subset R that satisfies these conditions for fixed constant multiples is called a "good" sample and is called "bad" otherwise. Clarkson's results show that by using a straightforward random sampling technique any randomly chosen subset is good with constant probability; implying that it can also be "bad" with constant probability. Consequently, his methods yielded *expected* resource bounds but cannot be used to obtain high-likelihood bounds (i.e., bounds that hold with probability $1 - 1/n^\alpha$ for any $\alpha > 0$). This makes it very difficult to extend his methods in the context of parallel algorithms due to the recursive nature of the algorithms. In particular, the *expected* bounds at each recursive call are not strong enough to bound the resources used by the entire parallel algorithm for the following reason. In a sequential algorithm, because of the linearity property of expectation (i.e., the expectation of the sum is the sum of expectations), it suffices to bound the expected time required by individual steps. The total expected time of the sequential algorithm is the sum of expected time of the individual steps. In contrast, consider the recursive parallel algorithm as a tree where a node corresponds to a procedure and the children of a node corresponds to the parallel recursive calls made by the procedure. The time required at each level of this tree is the *maximum* of the time required by any node of that level. There is no known method to bound the maximum of the expectations without using higher moments. The total time required by the parallel algorithm is the time when all the procedures corresponding to the leaf nodes are completed. Typically, in a parallel algorithm, the number of leaves in the corresponding process tree is at least n^ϵ $(0 < \epsilon < 1)$. Even if we succeed in bounding the expected time for completion of a leaf-node procedure, the expected bounds are too weak to bound the *maximum* of the time required by all such processes.

The above problem can be dealt with by developing a technique for choosing samples that are "good" (as defined above) with high probability. By doing so we shall show that a leaf-node process terminates in a given time bound with probability $1–1/n^\alpha$ for any $\alpha > 0$. In particular, for $\alpha > 1$, this implies that the failure probability for the entire algorithm is less than $1/n^{\alpha-1}$ (since there can be at most $O(n)$ leaf-level processes). We introduce a technique called *polling* to obtain a "good" sample with high probability with relatively small overhead. Roughly speaking, we choose a number, $p(n)$, of random subsets (typically $p(n) = O(\log n)$) independently. We then determine which of these subsets is "good" and we show with high probability that one of them is "good." This scheme, though effective, is not very efficient since we have to repeat the procedure $p(n)$ times. However, we show that we can draw conclusions about the "goodness" of a sample very accurately by using only a fraction (typically $1/p(n)$) of the input which then makes the polling scheme very efficient. This is actually very similar to the idea of polling a small fraction of the population to find out how the entire population would behave. This turns out to be crucial in bounding the total running time of the parallel algorithm. In addition to the applications in obtaining the improved results in this paper in computational geometry, polling appears to be a general tool for obtaining improved parallel randomized algorithms. A similar idea had been used previously by Dyer and Frieze [18] to improve the efficiency of a linear programming algorithm.

Note that the second property of a "good" sample, i.e., that of bounding the total size of the subproblems is not an issue in one-dimensional problems. In the parallel sorting algorithms of Reischuk [30] and Flashsort [19] the total size of the subproblems always equals the input size. This is another reason why the straightforward random sampling techniques do not carry over to the recursive algorithms. Clarkson [8] circumvents this problem by limiting the number of recursive levels by a fixed constant. By using recursion

over $s(n)$ steps the problem size could grow by a multiplicative factor of $2^{\Omega(s(n))}$ if the sum of the subproblems increases by only a constant factor over the parent-problem at every recursive call. This could seriously affect the efficiency of the algorithms, especially when we are looking for optimal algorithms. We need additional arguments to bound the total size of the subproblems at any level of recursive calls (independent of the level number).

1.3. Main results. The main result in this paper can be summarized as follows.

THEOREM 1.1. *There exists a randomized algorithm in the* CREW PRAM *model for constructing the intersection of n half spaces in three dimensions that runs in* $O(\log n)$ *time for any input with probability* $> 1 - 1/n^{\alpha}$ *(for any fixed* $\alpha > 0$*) using n processors. Moreover, we can also limit the total number of random bits used by our algorithm to* $O(\log^2 n)$.

The above theorem immediately implies equally efficient algorithms for the following problems from well-known reductions:

(i) Convex hull of points in three-dimensions,
(ii) Voronoi diagram of point sites in a plane,
(iii) Euclidean minimal spanning tree.

The previously best-known algorithms for all these problems are suboptimal by at least an $O(\log n)$ factor in time complexity. For the last problem we require a Priority CRCW PRAM model. In this model, the highest priority (fixed in advance) processor among any group of contending processors succeeds in the event of a write conflict.

We adopt a top-down approach in describing the algorithm. In §2 we list some of the preliminary results that will be used as low-level procedures in the algorithm and some probabilistic notations used to aid the analysis. In §3 we sketch a very high-level description of the algorithm that uses the straightforward random sampling (without polling) and which if implemented in a straightforward manner would not be very efficient. In §4 we give a formal description of polling and its probabilistic analysis. In §5 we describe an efficient procedure for for carrying out the divide step of the algorithm. In §6 we present probabilistic arguments for bounding the total time of the algorithm with high likelihood and bound the number of processors needed at any single step to complete the analysis.

2. Some preliminary results and overview.

2.1. Model of computation and notations. Throughout this paper we will be using the CREW PRAM model which is the synchronous shared memory model of parallel computation in which processors may simultaneously read from a memory location but are not allowed to write concurrently. At each step, a processor is allowed to perform a real-arithmetic operation consistent with standard models used for sequential geometric algorithms. Moreover, each processor has access to a random-number generator that returns in unit time a truly random number of $O(\log n)$ bits. However, see §6.3, where we limit the use of truly random bits.

The term *very high likelihood* (*probability*) is used in this paper to denote probability $> 1 - n^{-\alpha}$ for some $\alpha > 1$ where n is the input size. Just as the big-O function serves to represent the complexity bounds of deterministic algorithms, we shall use $\tilde{O}$ to represent complexity bounds of the randomized algorithms. We say that a randomized algorithm has resource bound $\tilde{O}(f(n))$ if there is a constant c such that the resource used by the algorithm is no more than $c\alpha f(n)$ with probability $\geq 1 - 1/n^{\alpha}$ for any $\alpha > 1$. (An equivalent definition will be bounding the resource by $\alpha \cdot f(n)$ with probability greater than 1 - $n^{-c\alpha}$, and in the rest of the paper they will be used interchangeably.) Note that an algorithm whose *expected* resource bound is $O(f(n))$ does not have any better confidence interval beyond using Markov's inequality, i.e., the probability that it exceeds

the resource bound by a factor k is less than $1/k$. This implies that the failure probability does not diminish as rapidly as the high-likelihood bounds. High-likelihood bounds are especially useful for parallel algorithms, where we need to bound the time complexity of all the processes. In contrast, the expected bounds as used by Clarkson [8] are difficult to use to bound the overall maximum time for all processes.

We will be using the term high-likelihood in a variety of situations throughout this paper that may look different from the canonical form given in the previous paragraph. We illustrate this with two lemmas which will be of use later.

LEMMA 2.1. *The union of k events (k being any fixed integer), each of which succeeds with high probability, also succeeds with high probability.*

If the failure probability of event i is less than $1/n^{\alpha_i}$ the failure probability of the union of the events is less than $\sum_{i=1}^{k} n^{-\alpha_i} < k/n^{\alpha}$, where $\alpha = \min(\alpha_1, \cdots, \alpha_k)$. This is less than $n^{-(\alpha-\delta)}$ for any $\delta > 0$. Note that the above holds with appropriate change of constants even if k is a polynomial in n of some fixed degree.

In the remainder of this paper we shall often refer to the logical structure of the recursive parallel algorithm as a *process tree* associated with the algorithm. The root of this tree is an instance of the original problem. Each internal node corresponds to a procedure and the children of a node corresponds to the parallel recursive calls made by the procedure.

THEOREM 2.1. *Given a process tree that has the property that a procedure at depth i from the root takes time T_i such that*

$$P[T_i \geq k\alpha \log n(\epsilon_0)^i] \leq 2^{-(\epsilon_0)^i \alpha \log n},$$

then all the leaf-level procedures are completed in $\tilde{O}(\log n)$ time. Here i denotes the distance of the procedure from the root level; k and α are constants greater than zero, and $0 < \epsilon_0 < 1$.

A rigorous proof of this assertion can be found in Reif and Sen [27] (a generalization of Reif and Valiant [29]). Intuitively, the time taken at a node which is at a distance i from the node is $O(\log n/2^i)$ with high probability.

For the rest of the paper, we assume that the success probability required by the algorithm is given, so that given n, we can fix α. From this, we can compute the required probability of success at every individual step of the algorithm even though we do not provide explicit formulae. Also, for convenience of notation, functions of n that may not be necessarily integral valued like $\log\log n$ or n^{ϵ} will actually denote the ceiling of such values, i.e., $\lceil \log\log n \rceil$ and $\lceil n^{\epsilon} \rceil$. This does not affect the asymptotic bounds of the algorithm.

2.2. Useful results. In the remainder of the section we shall assume that the half spaces are described as inequalities of the form $a_i x + b_i y + c_i z + d_i \geq 0$. We shall also use the terms "half-space" and its bounding "plane" interchangeably where it is clear from the context. The output is a list of vertices of the polyhedron C which is the intersection of the half spaces. The vertices are defined by the 3-tuples of the three intersecting planes defining the vertex. We assume that the planes are in general position, that is, every vertex is the intersection of exactly three planes. This assumption is for the convenience of analysis and not a real bottleneck for the algorithm. There are standard perturbation techniques that simulate the nondegeneracy condition. For example, changing the coefficients by a sufficiently small randomly chosen real value satisfies the property of nondegeneracy with probability approaching 1. The textbook [19] discusses symbolic perturbation techniques that are deterministic but more expensive. The edges of this polytope are those pairs of vertices that have two common planes in the tuples.

A face is defined by all tuples that have one common plane. A tuple can be written in six ways (permutation of the three planes) and thus sorting them (all six possible representation of the vertices) would enable us to obtain the faces and edges as the necessary adjacency structure of the polytope (which is a planar graph).

The following observation is useful for constructing the intersection of a random subset of half spaces that is used to split up the problem evenly.

LEMMA 2.2. *The intersection of a given set of n half spaces can be computed in $O(\log n)$ time using n^4 processors in a* CREW PRAM *model.*

Proof. Assuming nondegeneracy (i.e., no four planes intersect at a common point), there are $O(n^3)$ candidate vertices for vertices of the convex hull (of the intersection). For each vertex, test whether it is a vertex of the convex polyhedron by checking whether it satisfies all the equations defining the half spaces. This can be done trivially in $O(\log n)$ time using $O(n)$ processors for each candidate point. Only the vertices would survive. Determine the faces of the convex polyhedron by identifying planes that contain three vertices of the intersection. The necessary adjacency structure can be constructed by an application of sorting. □

LEMMA 2.3. *Given a set of n half spaces, it is possible to compute their intersection in $O(\log^3 n)$ time using n processors in a* CREW PRAM *model.*

Proof. The proof follows immediately from Aggarwal et al. [1]. □

The previous result is used to stop the recursion at a level when the problem size is small (typically $O(\log^k n)$ for some integer k) and to solve the problem directly. Note that any polylog-time algorithm using a linear number of processors would again suffice for our purpose. At this stage the problem size is so small that using a suboptimal algorithm will not affect the asymptotic complexity of the algorithm.

2.3. The dual transform. The convex hull problem has a very interesting dual problem, namely, the intersection of half spaces. This dual transformation $\mathcal{D}$ maps a point in E^d to a nonvertical hyperplane in E^d and vice versa. Let $p = (\pi_1, \pi_2, \cdots, \pi_d)$ be a point in E^d. Then $\mathcal{D}(p)$ is the hyperplane $1 = \pi_1 x_1 + \pi_2 x_2 + \cdots \pi_d x_d$ and vice versa such that a hyperplane h not containing the origin is mapped to a point p for which $\mathcal{D}(p) = h$.

The transform $\mathcal{D}$ is extended to sets of points (hyperplanes) in a natural way. Let $\mathcal{P}$ be a convex polytope with nonempty interior int$\mathcal{P}$ and assume that the origin O is contained in $\mathcal{P}$. Then $\mathcal{D}(\mathcal{P})$ is an infinite set of hyperplanes which avoid some convex region around O. The dual of $\mathcal{P}$ is defined as

$$\bar{\mathcal{P}} = \text{closure}\left(\bigcap_{h \in \mathcal{D}(\mathcal{P})} h^{\text{pos}}\right)$$

where h^{pos} denotes the half space containing the origin. The following observation can be verified [19].

LEMMA 2.4. *Point p belongs to the* int$\mathcal{P}$, boundary$\mathcal{P}$ *or* complement$\mathcal{P}$ *if and only if the hyperplane $\mathcal{D}(p)$ avoids $\bar{\mathcal{P}}$, avoids* int$\bar{\mathcal{P}}$ *but not $\bar{\mathcal{P}}$, or intersects* int$\bar{\mathcal{P}}$, *respectively.*

In other words, given a set of points S, the vertices of the convex hull are the dual transform of the facets of the intersection of the half spaces $\mathcal{D}(S)$ (which will be denoted by S^* in future). This property has been exploited very often so that the same algorithm can be used for both convex hulls and intersection of half spaces (if we know an interior point). In this paper we actually derive an algorithm for constructing the intersection of half spaces. Moreover, the dual transform has nice applications for searching (as used in §5).

3. A naive random sampling algorithm and its shortcomings. Before we embark on a formal proof of the main theorem, let us give an informal description of the algorithm using the straightforward random sampling strategy (as used by Clarkson [10]). We intentionally leave out polling from this preliminary discussion to illustrate the pitfalls of using naive sampling strategies for parallel algorithms. We shall assume for the time being that we know a point p^* in the intersection of the S^* and later show how to determine such a point efficiently. Using a random subset of S^*, we split the original problem *evenly* into smaller sized problems and then apply the algorithm recursively to each of the problems. By using a random subset of size n^ϵ, $(0 < \epsilon < 1)$ we split up the problem into subproblems of expected size $n^{1-\epsilon}$. This results in a recurrence of the form $\bar{T}(n) = \bar{T}(n^{1-\epsilon}) + f(n)$, where $f(n)$ is the time for dividing the problem. If $f(n) \leq \tilde{O}(\log n)$ (which requires the use of polling as described in §4), we have an algorithm whose expected running time is bounded by $O(\log n)$. We further need to show that the number of processors required at each step of the algorithm is $O(n)$.

ALGORITHM (*Main*).
Input: A set S^* of n half spaces $H_1, H_2, \cdots, H_n$.
Output: The output convex polyhedron $\mathcal{C}$, which is the intersection of the n half spaces.

(1) Choose a random subset $R \subset S$ of half spaces such that $|R| = n^\epsilon$ (for some ϵ, $0 < \epsilon < 1$ that we shall determine during the course of analysis).

(2) Find the intersection of the half spaces in R. Take a fixed plane and cut up each face of the polyhedron with (parallel) translates of this plane passing through the vertices. Thus each face is a trapezoid. Further, partition each trapezoid with a diagonal so that each face is triangular. For a face F_i consisting of vertices x_i, y_i, z_i consider the cone C_i formed by p^* as the apex and F_i as the base. Let C_R denote the number of cones. Note that $C_R \leq 2|R|$.

(3) For the remaining $S - R$ half spaces find the intersection of the planes (bounding these half spaces) with the cones. Note that a plane may intersect more than one cone. The intersection of the S half spaces is the union of the intersection of the half spaces intersecting a cone (over all cones). That is, $\mathcal{C}$ is $\cup_{i=1}^{C_R} I_i$ where I_i is $\cap_j \{H^j\}$ restricted to C_i.

(4) If the number of planes intersecting a cone is more than a predetermined threshold apply steps (1)–(3) recursively to this cone for the set of half spaces (bounded by the planes), or else solve the problem directly (using Lemma 2.3).

The algorithm outlined above that uses a straightforward random sampling in step (1) is only a skeleton of the actual algorithm and is not very efficient in its present form. One of the main problems is that in step (3) the total size of the subproblems could exceed the size of the parent (calling) problem by a large factor at each recursive call. Note that bounding this increase at each recursive call by a constant factor does not suffice. This would imply that after $O(\log\log n)$ levels, we can only bound the the total size of the subproblems at this stage by $O(n \log^{O(1)} n)$. This is where this algorithm differs from some other recursive parallel algorithms like randomized parallel sorting algorithms of [29], [30] where the total size of the subproblems is always bounded by the input size. We need more sophisticated methods for choosing the random subset in step (1) to prevent this. We will show in §4 how to solve this problem using polling. Moreover, coming up with a fast procedure for detecting the intersection of the half planes with the cones is in itself a nontrivial task. For the rest of the paper, we concentrate on individual steps and derive the necessary refinements to prove the main theorem.

4. Probabilistic lemmas.

4.1. The need for polling: An improved random sampling technique. A crucial part of the analysis rests on showing that a random subset R can be chosen efficiently in the first step of the algorithm that divides the problem into almost equal-sized subproblems. In addition, we have to show that the total size of the subproblems is the same as the complexity of the original problem at every stage of the recursive calls. The following result follows from Clarkson [10, Cor. 4.3] for any random $R \subset S$ with $|R| = r$.

LEMMA 4.1. *Let X_i denote the set of planes intersecting cone C_i (using the same terminology as in step* (2) *of the algorithm). Then the following conditions hold with probability at least* 1/2:

(i) $\sum_{i=1}^{C_R} |X_i| \leq k_{\text{total}}(n/r) \cdot E(C_R)$, *and*

(ii) $\max_i |X_i| \leq k_{\max}(n/r) \cdot \log r$,

where k_{total} and $k_{\max}$ are constants and C_R is defined previously.

Any subset of the input half spaces that satisfies the above conditions for some fixed constants is defined to be "good" and is otherwise "bad." A direct consequence of the lemma is that we can divide the problem into almost equal-sized subproblems, such that the increase in the original problem size can be bounded by at most a constant multiplicative factor of $k_{\max}$. Since our objective is to apply this recursively, we need a more sophisticated sampling algorithm to obtain a sample that is "good" with high likelihood.

4.2. An informal description of polling. The idea for choosing a "good" sample is as follows. Since the above events would fail only with constant probability, the probability that the conditions would fail in $O(\log n)$ independent trials is less than $1/n^\alpha$ for some $\alpha > 0$. Therefore, if we choose independently $p(n)(= O(\log n))$ sets of samples, one of them is good with very high likelihood. However, to determine if a sample is "good," we would have to carry out step (3) of the algorithm described in the previous section $O(\log n)$ times, each of which would require $O(\log n)$ time (such a method is described in §5). Instead, we try to estimate the the number of planes intersecting a cone C_i using only a fraction of the input planes. For example, we can choose $c_0 \cdot n/\log^d n$ half spaces for some fixed integer $d > 2$ and a constant c_0 of the input planes randomly for the jth sample R_j. The actual value of c_0 will be determined from the required success probability of the algorithm. Let X_i^j be the number of planes intersecting cone C_i corresponding to sample R_j, $1 \leq j \leq b \log n$, where b is fixed integer greater than zero. Let A_i^j be the number of planes intersecting C_i out of the $n/\log^d n$ randomly chosen input planes for the same sample. Clearly, A_i^j is a binomial random variable with parameters $c_0 \cdot n/\log^d n$ (total number of trials) and X_i^j/n (success probability in each trial). Assuming that X_i^j is greater than $\bar{c} \cdot \log^{d+1} n$, for some constant $\bar{c}$, we will apply Chernoff bounds (see appendix) to bound the estimates tightly within a constant multiplicative factor. Since we do it only for $1/\log^d n$ of the input planes, the total number of operations for the $O(\log n)$ random subsets is bounded by $O(n \log n)$ (as we show in the next section). Note that $X_i^j < \bar{c} \log^{d+1} n$ is an easy case since $n^\epsilon \cdot \bar{c} \log^{d+1} n$ for $\epsilon < 1$ is $o(n)$.

4.3. Probabilistic analysis of polling. More formally, by invoking Chernoff bounds (see Appendix equations (1) and (2)), for any $\alpha > 0$ (α is a function of c_0), there exists $c_1 > 0$, independent of n,

$$\text{Prob}(A_i^j \leq \alpha c_1 X_i^j/\log^d n) \leq 1/n^\alpha$$

and

$$\text{Prob}(A_i^j \geq c_2 \alpha c_0 \cdot X_i^j/\log^d n) < 1/n^{c_0 \alpha} < 1/n^\alpha$$

(for $c_0 > 1$). From the last two inequalities, X_i^j is bounded by $L^j = A_i^j \log^d n / c_0 c_2 \alpha$ from below, and $U^j =$ by $A_i^j \log^d n / c_1 \alpha$ from above. With appropriate changes in the constants, this condition holds with high likelihood (as defined in §2.1) for all X_i^j simultaneously. We do the procedure (described in the next section) simultaneously for all the samples R_j and choose the sample R_{j_o} using the following simple test.

ALGORITHM *(Polling).*
(Let $N^j = \sum A_i^j$ and the let actual number of intersections be T^j. Let E^j be our estimate for the sample j; upper and lower bounds obtained from N^j are denoted by U^j and L^j, respectively)

If $k_{\text{total}} n > U^j$ then accept sample R_j (since $k_{\text{total}} n \geq U^j \geq T^j$),

else if $k_{\text{total}} n \leq L^j$ then the sample is "bad" (since $k_{\text{total}} n \leq L^j \leq T^j$),

else if $L^j \leq k_{\text{total}} n \leq U^j$, then accept the sample R_{j_o} for which E^{j_o} is minimum. Since both $k_{\text{total}} n$ and T^{j_o} lie in this interval this guarantees that $T^{j_o} \leq c_3 \cdot k_{\text{total}} n$ where $c_3 = U^j / L^j$, which is a constant.

Recall, that from our earlier discussion that at least one of the samples would satisfy conditions 1 or 3 with very high likelihood. We summarize as follows.

LEMMA 4.2 (Polling lemma). *If we can choose a set of random splitters that expects to be "good," then by using the* **polling** *algorithm, as described earlier we obtain a sample that is "good" with high probability.*

The above procedure can actually be used in a more general situation where we need "good" samples with very high likelihood from samples that only expect to be "good." Moreover, according to our previous discussion, the extra amount of overhead does not affect the asymptotic work done by the algorithm, because it uses only a fraction of the input to test the samples.

5. Finding intersections quickly.

5.1. A locus-based approach for finding intersections. We now focus on a procedure to find the intersection of planes with each of the cones C_i. Notice that a plane may intersect more than one cone, which rules out detecting the intersections sequentially. That is, if a plane intersects n^δ cones ($\delta > 0$), we cannot afford to detect them one after the other since we are looking for an $O(\log n)$ time procedure. Note that in the sequential case, Clarkson and Shor's [11] randomized incremental constructions give optimal expected time bounds for computing the plane-cone intersections that cannot be applied in our case.

We shall use a *locus-based* approach to solve this problem. This approach involves considering each query as a higher-dimensional point and partitioning the underlying space into regions providing the same answer. Thus any query problem can be reduced to a point location problem given sufficient preprocessing time and space. In our case, we have to preprocess the convex polytope of the sampled half spaces in such a way that given any plane, we should be able to report the list of cones that it intersects in $O(\log n)$ time using at most k processors where k is the number of intersections. We shall show that the preprocessing for a convex polytope of size n can be done in $O(\log n)$ parallel time using $O(n^c)$ processors, where c is a fixed constant. Thus we can choose any sample of size less than $n^{1/c}$ since we have n processors. For our problem, the a value of c is will be worked out in the proof of Lemma 5.1.

Given a convex polytope in three dimensions of size $O(n)$ along with an internal point which is the apex of the cones, there are only a polynomial (in n) number of combinatorially distinct possibilities of the way any given plane can intersect the cones. This can be seen from the following simple argument. Given any plane that intersects the polyhedron, we can perturb the plane without changing the cones it intersects so long as it remains within a fixed set of bounding vertices. Figure 1 illustrates the situation for a two-dimensional case. If we consider an equivalence relation where two lines are equivalent if and only if they intersect the same sets of cones, then the equivalence classes correspond to the cells in the arrangement $\mathcal{A}(H)$ where $H = \{\mathcal{D}(p) : p$ is a vertex of the convex n-gon or internal point and $\mathcal{D}$ is a dual transform$\}$ (see [19] for more details). Given any query line l, the cones that it intersects is defined by the partition of $\mathcal{A}(H)$ that $\mathcal{D}(l)$ belongs to. This observation can be extended to hold for any dimension; in our case dimension 3. If we consider the partitions of the 3-space induced by the intersections of the constraining half spaces, these are equivalent classes with respect to the cones they intersect. Notice that even if this partitioning may not be minimal it suffices for our purpose. All that remains to be done is to precompute for each of these regions the cones that the corresponding planes would intersect so that for any query plane in the same equivalence class we can list the intersecting planes in a table.

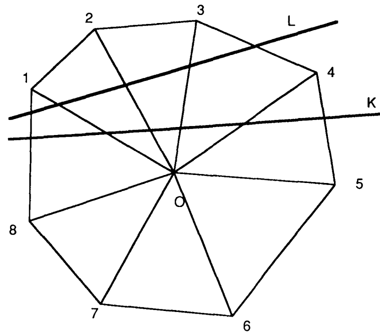


FIG. 1. *Lines L and K intersect a different set of sectors. In the dual plane the duals of L and K lie on different faces—in this case separated by the dual of vertex* 4.

5.2. A point-location algorithm. For the point-location problem, we use a preprocessing scheme due to Dobkin and Lipton [17] because of the ease in parallelization. The following is a fairly straightforward extension of their method.

LEMMA 5.1. *Given a set of m planes in E^3, it can be preprocessed in $O(\log m)$ time using $O(m^7)$ processors, such that given an arbitrary query point, the unique cell containing that point can be reported in $O(\log m)$ time. The space required is $O(m^7)$.*

Proof. Find the pairwise intersections of the given set of planes (there are $O(m^2)$ of them). Project the resulting lines on a plane not normal to any of the lines. Find the pairwise intersections of the straight lines and consider their projection on the x-axis. There are $O(m^4)$ intervals induced by these. For each of these intervals, order the straight-lines in increasing ordinates by sorting. This can be done in $O(\log m)$ time using $O(m^2)$ processors for each of the $O(m^4)$ intervals. There are now $O(m^6)$ trapezoidal regions. For each of these, order the planes in increasing z-coordinates for binary-search by sorting which are totally ordered in these subdivisions. This can be done in $O(\log m)$ time using $O(m^7)$ processors. The cells induced by this preprocessing are homeomorphic to a 3-cube, so that given any query point it can be located in such a subdivision with three binary searches. $\square$

For each of the cells in 3-space, we can precompute the cones that the corresponding plane intersects using $O(n^8)$ processors (by choosing a representative point in each cell and testing it against all the cones). Note that these subdivisions are finer than the minimal equivalence classes, i.e., more than one subdivision could have the same set of intersecting cones. We also store the number of intersecting cones for each of the subdivisions so that while listing the number of cones each query plane intersects we can do the processor allocation easily in $O(\log n)$ time using a prefix computation. By choosing less than $n^{1/8}$ samples, we can complete the entire preprocessing in the required time and processor bounds.

We summarize our conclusion in this section as follows.

LEMMA 5.2. *Step* (3) *of the Main Algorithm uses* $O(n)$ *space and terminates in* $O(\log n)$ *time using n processors in a* CREW PRAM *model.*

6. Controlling the size of subproblems and processor allocation. From Lemma 4.1, we know that the size of the problem can increase by a constant factor at each level and we wish to prevent this from happening over $O(\log \log n)$ levels, which would increase the number of processors required by a polylog factor. For this we need to quickly identify the redundant planes that do not contribute to the output complexity and to eliminate them from further recursive calls. This enables us to get a global bound on the total size of the subproblems (at any stage) which we shall show to be linear in the input plus the output size. More specifically, we allocate the processors recursively to the cones such that the number of processors is proportional to the number of output vertices in that cone, thereby bounding the number of processors to be $O(n)$. The following description provides more details of this scheme.

After we have found the planes intersecting a particular cone, we categorize them as follows:

(a) Planes that are completely occluded by another plane in the cone and hence these cannot be a part of the output in the cone,
(b) Planes that are occluded because of more than one other plane in the cone, i.e., there is no one plane that completely occludes them,
(c) Planes that contribute to an edge without an endpoint, i.e., the endpoints lie in some other cones,
(d) Planes that do contribute to a vertex in the cone.

To eliminate planes of type (a), we use a variant of the three-dimensional maxima algorithm. The three-dimensional maxima problem is defined as:

Given a set S of n points in a three-dimensional space, determine all points p in S such that no other point of S has x, y and z coordinates that simultaneously exceed the corresponding coordinates of p. In case there is such a point q, q is said to *dominate* p.

Since cones have a triangular base there are three edges that join it to the apex p^*. We sort the intersections of the planes with an edge at increasing distances from the apex. We repeat this for all the three edges. Call these three edges X, Y, Z and denote the intersection of a plane h_i as X_i , Y_i , Z_i and the ranks in the sorted list as $r(X_i)$, $r(Y_i)$, and $r(Z_i)$.

Observation 1. A plane A is occluded completely by another plane B if and only if it is dominated on its ranks of intersection on all the three edges by plane B.

This gives an effective strategy for eliminating planes of type (a) by identifying the complement of the set of the maximal elements, where we use the ranks of the intersection on the three edges as the order relation. Using the algorithm of [2], we can do this

in $O(\log n)$ time using a linear number of processors. For a two-dimensional illustration see Fig. 2.

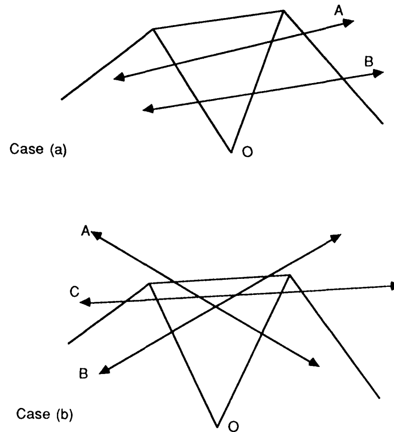


FIG. 2. (a) *A cannot show up in the output of the cone since it is "dominated" by B (on both edges of the cone).* (b) *C cannot be a part of the output but it is not dominated by A or B. So it cannot be eliminated by case* (a).

To identify planes of type (b), (c), and (d) we construct the intersection of the convex polytope C with each of the three faces of the cone. These are intersections of the faces with C that are two-dimensional convex polygonal chain. These will be referred to as *contours* in the following discussion. The contours can be computed in $O(\log n)$ time with n processors using any of the optimal two-dimensional convex hull algorithms. Note that these convex contours on the three faces are a part of the output and any plane that appears on this contour is a part of the final output. Consequently, a plane of type (b) cannot be a part of this contour. Unfortunately, there can be planes that are part of the output but are not part of any contour. Consider a plane that chops off a portion of the polytope within the cone. For the time being let us focus on only those planes that show up in the contours and consider the three-dimensional convex polyhedron formed only by these planes within a cone. We shall refer to such a three-dimensional polytope as a *skeletal hull.* We shall use the term "flattening" to imply that the vertices of the contour are projected along edges (intersection of two planes) they lie on, such that all of them become coplanar. Notice that there may be several such planes. We just choose one arbitrarily and these projected vertices defines a "base" face.We now make the following observation.

Observation 2. Any plane that is not a part of the contour on any face can intersect at most one skeletal hull.

This follows from convexity. Notice that such planes are not necessarily a part of the output but we are not aiming for an output sensitive algorithm. The previous observation guarantees that if a plane is not a part of C it will not survive in more that one cone when the algorithm is called recursively in the cones. The planes that do not intersect the skeletal hull cannot be a part of C within the cone.

A plane can be a part of the contour and not contribute to any vertex of the convex polytope C, that is, it only contributes an edge of the hull within the cone. In this case the edge intersects the cone in exactly two faces and the these vertices can be labeled by the two intersecting planes (which contributes to the output edge). Thus these planes can be identified quickly using sorting on the labels of the intersecting planes. We shall call these *free edges*.

The objective of the above procedure is to eliminate some of the planes that do not contribute to the output and ensure that going into any recursive call, the sum of the subproblems is less than the output size. We define the output size of the three-dimensional convex polytope to be $3|V|$ where V is the number of vertices of the convex polytope. Since the surface of the convex polytope is a connected planar map, we can use Euler's equation to show that $|V| = 2|F| - 2$ where F is the set of faces in the polytope. Since $|F| \leq n$, $|V| \leq 2n - 2$. This calculation is done using the property that each vertex is of degree 3 (which follows from our assumption that the planes are in general position). Let the number of processors be $6n$. We distribute the processors among the subproblems depending on the output size. For a cone C_i that does not contain any free edge, the output can be bound by the following claim.

CLAIM 6.1. *The output size of a cone is bounded by* $3n_i + 6m_i - 6$, *where* n_i *is the number of planes in the contour contributing at least one vertex and* m_i *is the number of planes of types* (b) *and* (d).

Proof. Let e_1 denote the number of edges of C that intersect the contour and contribute a vertex within the cone (the vertices of the contour are these edges). Let e_2 be the number of edges that lie within the cone (including both endpoints). Let v be the number of vertices of C within the cone (these have degree 3), then $e_1 + 2e_2 = 3v$ or

$$e_1 + e_2 = \frac{3v + n_i}{2}$$

as $e_1 = n_i$. Consider the planar map of the polyhedron formed by n_i and m_i planes and a "base" face by "flattening" the contour. Refer to the explanations of the terms "base" face and "flattening" in the previous paragraphs. The number of edges on the contour equals the number of vertices on the contour. Then by applying Euler's formula, $|V| = n_i + 2n_f - 2$ where n_f is the number of faces of of C that show up in the cone but are not a part of the contour. Since n_f (type (d)) can be bounded by m_i, the claim is proved. □

Notice that if a cone contains d free edges, then the above formula can be applied separately to each of the $d + 1$ partitions (induced by the free edges). The participant planes in each of these partitions are disjoint, giving us the following.

CLAIM 6.2. *The output size of a cone is bounded by* $3n_i + 6m_i - 6d$ *where* d *is the number of free edges.*

The processor allocation strategy is simply to allocate this number of processors to the subproblem (in the cone). The total number of vertices over all the cones is bound by the maximum output size, and by our allocation strategy we are allocating processors proportional to the maximum output size in each cone. Note that the actual output size may be less but we shall never have fewer processors than required, and this maximum size can actually be achieved. Another way to look at the processor allocation strategy is that two processors are allocated to each edge of the output hull and we allocate those processors to the cones which contain (or potentially contain) vertices associated with that edge. The free edges are not allocated any processors. Hence we have sufficient number of processors.

More formally, let us denote the edges with one endpoint and two endpoints in cone i by e_1^i and e_2^i, respectively. For the following discussion, a cone will refer to the portion of the cone (as previously defined) further split up by the free edges; that is, cone i in the new definition does not contain any free edge. This simplifies the discussion considerably without affecting our previous observations. From previous discussion and equation (1),

$$e_1^i + e_2^i = \frac{3v + n_i}{2}. \tag{1}$$

Using $e_1^i = n_i$,

$$e_2^i = n_i + 3q_i - 3, \tag{2}$$

where q_i is the number of planes in cone i that do not show up in the contour. If a total of N planes show up in the contour, the convex hull C_N corresponding to N planes has $3N$ edges from Euler's formula. Then

$$1/2\sum \hat{e}_1^i + \sum \hat{e}_2^i = 3N$$

$$\Rightarrow 1/2\sum n_i + \sum (n_i - 3) = 3N$$

using $q_i = 0$ in equation (2). Here the summation is over all r cones and $\hat{e}_1^i$ and $\hat{e}_2^i$ are restricted to edges of C_N. By simplifying we get

$$\sum n_i = 2N + 2r. \tag{3}$$

The total number of processors required is

$$2\sum e_2^i + \sum e_1^i \tag{4}$$

$$= 2\sum e_1^i + 6\sum q_i - 6r + e_1^i$$

from equation (2).

$$= 6\sum n_i + 6\sum q_i - 6r$$

$$= 6N + 6r + 6\sum q_i - 6r$$

from equation (3). Since $\sum q_i \leq n - N$, (from Observation 2, each plane in q_i is counted exactly once) the total number of processors is less than $6n$ from the previous equation. Since we started with $6n$ processors, we have sufficient number of processors for the recursive call. This argument can be applied inductively.

We shall now describe a procedure to construct the *skeletal hull* within a cone and preprocess the skeletal hull such that queries of the kind plane-polyhedra intersection detection can be answered quickly. The latter part can be done efficiently using a hierarchical polyhedra decomposition scheme due to Dobkin and Kirkpatrick [15]. The construction of the hierarchical representation can be done in $\tilde{O}(\log n)$ time using an algorithm of described in Reif and Sen [27] (also discovered independently by Dadoun and Kirkpatrick [14] but the analysis given in their paper is not sufficient for our purposes). Given this representation, the plane-polyhedra intersection detection query can be answered in $O(\log n)$ sequential time [16].

We shall now discuss how to construct the skeletal hulls quickly. Although the skeletal hulls are themselves three-dimensional convex polytopes they have a much simpler structure (see Fig. 3). More specifically, they have the following property: all faces are unbounded (i.e., they are part of the contours). This implies that, if we construct them recursively using the same algorithm, we do not have to worry about case (b) since all planes that are part of the output will show up in the contours, and this holds for any

level of the recursive call. From the analysis given in the next section the skeletal hulls can be constructed in $\tilde{O}(\log n)$ time using a linear number of processors. The reader should be convinced that there is no circularity of arguments here. One way to look at the problem is the following: Assuming that there are no planes which satisfies both cases (b) and (d) (i.e., all planes that are part of the output show up in the contours), the algorithm terminates in $\tilde{O}(\log n)$ time using a linear number of processors. So after having constructed the skeletal hull for the cone, the redundant planes are quickly eliminated using the procedure outlined in the previous paragraph. Subsequently, the algorithm is called recursively on the cone—this time to build the actual polytope C (as opposed to the skeletal hull).

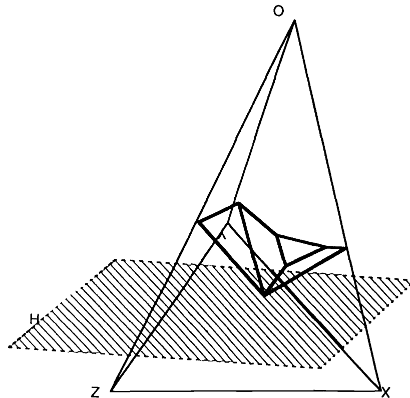


FIG. 3. *OXYZ is a cone from the sampled half spaces and the thickened lines show a skeletal hull in the cone. H is a half space that does not intersect the cone but is a part of the output within the cone.*

6.1. Final analysis. Consider the algorithm as a tree where each node corresponds to a procedure and the children of a node representing processes corresponding to the recursive calls made by the procedure. Then the running time of the algorithm corresponds to a worst-case sequence of nested procedure calls along any path in this tree from the root to a leaf node. Let us first analyze the cost of constructing the skeletal hull at any node. This process tree corresponding to this algorithm has the following property. If we choose the sample size to be $O(n^{1/8})$ a process at level i $(1 < i < O(\log\log n))$ has size less than $O(n^{(9/10)^i})$ (from Lemma 4.1) and the process terminates in time $O(\log n^{(9/10)^i}) = (9/10)^i O(\log n)$ with probability greater than 1–$1/n^{(9/10)^i}$. From Theorem 2.1, any nested sequence of recursive calls exceeds time $c\gamma \log n$ with probability less than $1/n^{\gamma}$ for any $\gamma > 1$. It follows that all the leaf processes and hence the algorithm is completed within the same time with high likelihood. We may may thus conclude that the skeletal hull at any node with problem size n_i can be constructed in $\tilde{O}(\log n_i)$ time.

For the overall algorithm, we add the time for detecting redundant half planes. From the previous section this involves constructing a data structure for point location for detecting the plane-polyhedron intersections. Since this also takes $\tilde{O}(\log n_i)$ time, from Lemma 2.1, the total time is also $\tilde{O}(\log n_i)$ (with appropriate constants in the $\tilde{O}$ notation). Another appeal to Theorem 2.1 results in the $\tilde{O}(\log n)$ running time for the overall algorithm. The space used is $O(n)$ at step 3 of each recursive level giving a total bound of $O(n \log\log n)$ for all the $O(\log\log n)$ recursive levels of the algorithm. This proves the main result of the section.

COROLLARY 6.1. *The following problems can be solved in* $\tilde{O}(\log n)$ *time using* n *processors in the* CREW PRAM *model:*

(i) *Convex hull of a set of points in three-dimensions,*

(ii) *Voronoi diagram of point-sites in plane,*

(iii) *All-points nearest neighbor,*

(iv) *Euclidean minimal spanning tree.*

The algorithm for the last problem uses concurrent-write feature where as the remaining algorithms run on CREW *model.*

Proof. (i) follows immediately because of well-known reduction of convex hulls to intersection of half spaces. To determine an internal point p^* in the intersection, we can determine an internal point of the convex hull and use it as the origin for the duality transform. The origin is known to be contained in the intersection of the half spaces.

For (ii), given a set of n points in the plane we apply an inversive transformation given by Brown [4] to the input points to transform the problem into finding the convex hull of n points in 3-space.

(iii) can be obtained in $O(\log n)$ time from the Voronoi diagram.

(iv) can be obtained by running a minimal-spanning tree algorithm on the edges of Delaunay triangulation which is the dual graph of the Voronoi diagram. This algorithm uses the stronger Priority CRCW model as in [3]. □

7. Bounding random bits.

7.1. Chebychev's inequality. The commonly used form of Chebychev's inequality has the form

$$\text{Prob}[(|X-\mu| \geq t)] \leq \frac{\sigma^2}{t^2}.$$

This simple fact was exploited by Chor and Goldreich [5] for their two-point sampling theorem where they consider the following scenario. To determine if a property P holds for x (for example if x is prime), we often use a function $f(x,r)$ which satisfies the condition that if P holds for x then $f(x,r)$ is 1 with probability at least 1/2 whereas if P does not hold for x then $f(x,r)$ is 0. Here r is a witness which is a random number in a certain range. This implies that if $f(x,r)$ is 1, then repeating the experiment for t *independent* random witnesses decreases the failure probability (of incorrectly categorizing x) to 2^{-t}. Instead of choosing t independent witnesses, if we choose t witnesses that are pairwise independent then we can analyze the probability of error as follows. Assume that in t trials, $f(x,r_i)$ is zero for all the trials and $P(x)$ is true. Let $Y=\sum_i f(x,r_i)$. Then $E[Y] \geq t/2$ and the variance $\sigma_Y \leq \sqrt{t}/2$. Then $\text{Prob}[Y=0] \leq \text{Prob}[|Y-E[Y]|] \geq t/2$ which is less than $1/t$ from Chebychev's inequality.

The t *pseudorandom* numbers can be generated from two purely random numbers by using the scheme $r_i = a + bi$ where a and b are random numbers. Thus instead of the confidence bound of 1/4 (for two numbers), we can do much better, i.e., $1/t$. Karloff and Raghavan [23] were able to extend these techniques to show that Reischuk's sorting algorithm can be implemented in the same asymptotic bounds by using only $O(\log n \log\log n)$ purely random bits (instead of the naive scheme requiring $O(\sqrt{n})$ random bits). Here we generalize their scheme to minimize the number of random bits used by the algorithms described in the previous chapter. For this we need to prove some preliminary results.

DEFINITION. A family of random variables is called k-way independent if any subset of k variables are mutually independent.

Clearly a k-way independent family is l-way independent for any $l \leq k$.

Let p be a suitable prime number and choose k numbers a_i, $0 \le i \le k-1$ randomly from Z_p. Consider the numbers of the form $r_i = (\sum_{j=0}^{j=k-1} a_j \cdot i^j) \bmod p$. The following is a well-known result [5] for this class of pseudorandom number generators.

Fact 1. The numbers r_i's are uniformly distributed in Z_p and are also k-way independent.

Fact 2. If X_i, $1 \le i \le n$ are n mutually independent random variables and ϕ_s are real-valued functions, then

$$E\left[\prod_{s=1}^{n} \phi_s(X_s)\right] = \prod_{s=1}^{n} E[\phi_s(X_s)].$$

LEMMA 7.1 (Generalized Chebychev inequality).

$$\text{Prob}\{|X| \ge t\} \le \frac{E(\phi(X))}{\phi(t)}$$

where ϕ is a positive monotonically increasing function.

Proof. Let $\text{Prob}[X = x_j] = f(x_j)$. Then

$$\begin{aligned}\text{Prob}\{|X| \ge t\} &= \sum_{|x_j| \ge t} f(x_j) \\ &\le \sum \frac{\phi(|x_j|) f(x_j)}{\phi(t)} \\ &\le \frac{E(\phi(|X|))}{\phi(t)}.\end{aligned}$$

LEMMA 7.2. *Let X be the sum of n $2k$-way independent and identical Bernoulli random variables X_i, $1 \le i \le n$, each of which has a success probability p. Then for a fixed k (chosen independently of n),* $\text{Prob}\{|X - \mu| \ge \mu\} \le O(\frac{1}{\mu^k})$ *where $\mu = np$ and $p \le O(n^{-\beta})$ for some $0 < \beta < 1$.*

See appendix for proof.

COROLLARY 7.1.

$$\text{Prob}\{|X - \mu| \ge a \cdot \mu\} \le O\left(\frac{1}{a^k \cdot \mu^k}\right)$$

where a is a constant between zero and 1.

Proof. Substitute $t = a\mu$ in the previous lemma. □

7.2. Rederiving probabilistic bounds with fewer random bits. In order to limit the number of random bits, we shall rederive some of the random-sampling bounds and the *polling lemma* using the $2k$-way independent random variables. In particular, we shall prove a slightly weaker bound than Lemma 4.1.

LEMMA 7.3. *The probability that the maximum number of half planes intersecting any cone exceeds $n^{1-\epsilon+\delta}$ is less than $n^{-k\delta+5}$. Here the size of the sample is n^ϵ and $0 < \delta < \epsilon < 1$ for some constants ϵ and δ.*

Proof. By randomly choosing n^ϵ half spaces, the expected number of half spaces chosen in the sample in a sector that has more than $n^{1-\epsilon+\delta}$ half planes is greater than n^δ. Thus the probability that none of the half planes was chosen in the sample is less than $n^{-k\delta}$ from Lemma 7.2. Summing over the $O(n^5)$ possible cones (see Clarkson [10] for justification of this bound) gives us the required result. □

Although this bound is somewhat weaker than Lemma 4.1, it still suffices to show that the process-tree (corresponding to the algorithm) has $O(\log\log n)$ depth (the constant is somewhat larger). Condition (i) of Lemma 4.1 is not affected since its proof in [10] uses only expectations.[1] For probabilistic analysis of polling we make use of the fact that if the expected number of half-planes in a sector is larger than n^β then Lemma 7.3 directly yields high probability bounds for some $0 < \beta < 1$ and choosing a sufficiently large value for k. If the mean is less than n^β, then for small β, $n^{\epsilon+\beta} = o(n)$.

An identical argument can be carried out for the problems tackled in Reif and Sen [27], namely, the trapezoidal decomposition. Notice that the Chernoff bounds that we were using earlier yielded much stronger bounds (of the order of 2^{-n^ϵ}) which was not required to prove our polling lemma. The ramifications of using these bounds are that the constants associated with the running time for the same confidence bounds are much larger.

From here on we can directly use the scheme of Karloff and Raghavan. The random bits can be shared by the $O(n)$ paths. We need an extra $O(\log n)$ multiplicative factor of truly random bits for implementing polling for which we need $O(\log n)$ independently chosen $2k$ random seeds of $O(\log n)$ bits each. Moreover, the point location algorithm can be shown to require only $O(\log n)$ bits (Sen [31]). We summarize as follows.

THEOREM 7.1. *The algorithm for constructing three-dimensional convex hulls runs in* $\tilde{O}(\log n)$ *time using n processors and* $\tilde{O}(\log^2 n \log\log n)$ *purely random bits.*

8. Concluding remarks. The randomized algorithms presented here reinforce the optimism expressed in an earlier paper (Reif and Sen [27]) where we introduced randomization as an effective tool for developing parallel algorithms in computational geometry. Clarkson had demonstrated the usefulness of randomization for deriving improved expected time bounds for a large number of sequential algorithms. Although we draw from Clarkson's work, our results should be of independent interest because of many unique additional difficulties presented by the parallel environment and the techniques needed to tackle them.

This paper describes the first $O(\log n)$ parallel time algorithm with optimal speed-up for three-dimensional convex hulls and related problems; however, a number of questions are left unanswered. The most obvious problem is that of designing a deterministic algorithm with same bounds. It is possible that an optimal algorithm for two-dimensional Voronoi diagrams may be easier to obtain than a similar algorithm for three-dimensional convex hulls. Moreover, we use the CREW PRAM model for our algorithm, raising the question of whether the algorithm can be made to run without the feature of concurrent reads. A more theoretical issue is that of designing sublogarithmic time parallel algorithms for all these problems with optimal speed-ups. Also can the probabilistic bounds be improved from $1 - 1/n^c$ for any c to say, $1 - 2^{-n}$?

An extremely important area of investigation in the field of parallel algorithms for computational geometry is development of efficient algorithms for fixed interconnection networks like hypercubes and butterfly networks. In spite of some elegant work done in the PRAM model, the currently best-known results for almost all these fundamental problems except two-dimensional convex hulls remain suboptimal. It appears very unlikely that the optimal algorithms would be deterministic since there are no known optimal deterministic sorting algorithms for these networks. This should encourage more research in the area of developing more sophisticated probabilistic methods for parallel computational geometry. Recently, Reif and Sen [28] were able to make some progress

[1] Clarkson [7] pointed it out to the authors.

in this direction by presenting efficient algorithms for triangulation.

A. Appendix. We say a random variable X upper-bounds another random variable Y (equivalently Y lower bounds X) if for all x such that $0 \leq x \leq 1$, $\text{Prob}(X \leq x) \leq \text{Prob}(Y \leq x)$.

A Bernoulli trial is an experiment with two possible outcomes viz. success and failure. The probability of success is p.

A binomial variable X with parameters (n, p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p. The *probability mass function* of X can be easily seen to be

$$\text{Prob}(X \leq x) = \sum_{k=0}^{x} \binom{n}{k} p^k (1-p)^{n-k}.$$

The tail end of the binomial distribution can be bounded by *Chernoff* bounds. In particular, the following approximations due to Angluin and Valiant are frequently used:

$$\text{Prob}(X \geq m) \leq \left(\frac{np}{m}\right)^m e^{m-np}, \tag{1}$$

$$\text{Prob}(X \leq m) \leq \left(\frac{np}{m}\right)^m e^{-np+m}, \tag{2}$$

$$\text{Prob}(X \leq (1-\epsilon)pn) \leq \exp(-\epsilon^2 np/2), \tag{3}$$

$$\text{Prob}(X \geq (1+\epsilon)np) \leq \exp(-\epsilon^2 np/3) \tag{4}$$

for all $0 < \epsilon < 1$. The last two bounds actually follow from the Chernoff bounds which (for a discrete distribution) can be stated as

$$\text{Prob}[A \geq x] \leq z^{-x} G_A(z)$$

where $G_A(z)$ is the probability generating function. To minimize the bound we substitute $z = z_o$ that minimizes the right side expression.

REFERENCES

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap, *Parallel computational geometry*, Proc. of 25th Annual Symposium on Foundations of Computer Science, pp. 468–477, 1985; also in Algorithmica, 3 (1988), pp. 293–327.

[2] M. J. Atallah, R. Cole, and M. T. Goodrich, *Cascading divide-and-conquer: A technique for designing parallel algorithms*, SIAM J. Comput., 18 (1989), pp. 499–532.

[3] B. Awerbuch and Y. Shiloach, *New connectivity and msf algorithms for ultracomputer and pram*, Proc. of the International Conference on Parallel Processing, 1983, pp. 175–179.

[4] K. Q. Brown, *Voronoi diagram from convex hulls*, Inform. Process Lett., 9, pp. 223–228.

[5] B. Chor and O. Goldreich, *On the power of two-point sampling*, J. Complexity, 5 (1989), pp. 96–106.

[6] A. Chow, *Parallel algorithms for geometric problems*, Ph.D. thesis, University of Illinois, Urbana-Champaign, IL, 1980.

[7] K. Clarkson, personal communication.

[8] K. L. Clarkson, *A probabilistic algorithm for the post-office problem*, Proc. of the 17th Annual SIGACT Symposium, 1985, pp. 174–184.

[9] ———, *New applications of random sampling in computational geometry*, Discrete Comput. Geom., (1987), pp. 195–222.

[10] ———, *Applications of random sampling in computational geometry* II, Proc. of the 4th Annual ACM Symposium on Computational Geometry, 1988, pp. 1–11.

[11] K. L. Clarkson and P. Shor, *Algorithms for diametral pairs and convex hulls that are optimal, randomized and incremental*, Proc. of the 4th ACM Symposium on Computational Geometry, 1988.

[12] R. COLE, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.
[13] R. COLE AND M. T. GOODRICH, *Optimal parallel algorithms for polygon and point-set problems*, Proc. of the 4th ACM Symposium on Computational Geometry, 1988, pp. 201–210.
[14] N. DADOUN AND D. G. KIRKPATRICK, *Parallel processing for efficient subdivision search*, Proc. of the 3rd Annual ACM Symposium on Computational Geometry, 1987, pp. 205–214.
[15] D. DOBKIN AND D. KIRKPATRICK, *A linear time algorithm for determining the separation of convex polyhedra*, J. Algorithms, 6 (1985), pp. 381–392.
[16] ———, *Determining the separation of preprocessed polyhedra—a unified approach*, in Proc. International Colloquium on Automata, Languages, and Programming, M. S. Paterson, ed., Lecture Notes in Computer Science, Springer-Verlag, 1990, pp. 400–413.
[17] D. DOBKIN AND R. J. LIPTON, *Multidimensional searching problems*, SIAM J. Comput., 5 (1976), pp. 181–186.
[18] M. E. DYER AND A. FRIEZE, *A randomized algorithm for fixed-dimensional linear programming*, unpublished manuscript, 1987.
[19] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, Berlin, New York, 1987.
[20] S. FORTUNE, *A sweepline algorithm for Voronoi diagrams*, Algorithmica, 2 (1987), pp. 153–174.
[21] H. GAZIT, *An optimal randomized parallel algorithm for finding connected components in a graph*, Proc. of the 27th Annual IEEE Symposium on Foundations of Computer Science, 1986, pp. 492–501.
[22] D. HAUSSLER AND E. WELZL, *ϵ-nets and simplex range queries*, Discrete Comput. Geom., 2 (1987), pp. 127–152.
[23] H. KARLOFF AND P. RAGHAVAN, *Randomized algorithms and pseudorandom numbers*, Proc. of the 20th Annual STOC, 1988.
[24] C. LEVCOPOULOS, J. KATAJAINEN, AND A. LINGAS, *An optimal expected-time parallel algorithm for Voronoi diagrams*, Scandinavian Conference on Theoretical Computer Science, 1988.
[25] K. MULMULEY, *A fast planar partition algorithm* 1, Proc. of the 29th IEEE FOCS, 1988, pp. 580–589.
[26] M. GOODRICH, R. COLE, AND C. DUNLAING, *Merging free trees in parallel for efficient Voronoi diagram construction*, Proc. of the ICALP, 1990, pp. 432–445.
[27] J. H. REIF AND S. SEN, *Optimal randomized parallel algorithms for computational geometry*, Proc. 16th International Conference on Parallel Processing, 1987. Revised version, Duke University Tech. Report CS-88-01.
[28] ———, *Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications*, Proc. of the 2nd Annual Symposium on Parallel Algorithms and Architectures, 1990, pp. 327–337.
[29] J. H. REIF AND L. G. VALIANT, *A logarithmic time sort for linear size networks*, J. ACM, 34 (1987), pp. 60–76.
[30] R. REISCHUK, *A fast probabilistic parallel sorting algorithm*, Proc. of the 22nd IEEE Foundations of Computer Science, 1981, pp. 212–219.
[31] S. SEN, *Random Sampling Techniques for Efficient Parallel Algorithms in Computational Geometry*, Ph.D. thesis, Duke University, Durham, NC, 1989.
[32] M. SHAMOS AND D. HOEY, *Closest-point problems*, Proc. of the 7th ACM STOC, pp. 224–233.
[33] L. G. VALIANT, *A scheme for fast parallel communication*, SIAM J. Comput., 11 (1982), pp. 350–361.

SIAM J. COMPUT.
Vol. 21, No. 3, pp. 486–506, June 1992
© 1992 Society for Industrial and Applied Mathematics
005

PROCESSOR EFFICIENT PARALLEL ALGORITHMS FOR THE TWO DISJOINT PATHS PROBLEM AND FOR FINDING A KURATOWSKI HOMEOMORPH*

SAMIR KHULLER†, STEPHEN G. MITCHELL‡, AND VIJAY V. VAZIRANI§

Abstract. The authors give a parallel algorithm for finding vertex disjoint s_1, t_1 and s_2, t_2 paths in an undirected graph G. An important step in solving the general problem is solving the planar case. A new structural property yields the parallelization, as well as a simpler linear-time sequential algorithm for this case. The algorithm is extended to the nonplanar case by giving a parallel algorithm for finding a Kuratowski homeomorph, and, in particular, a homeomorph of $K_{3,3}$, in a nonplanar graph. The algorithms are processor efficient; in each case, the processor-time product of the algorithms is within a polylogarithmic factor of the best-known sequential algorithm.

Key words. parallel algorithms, disjoint paths, Kuratowski homeomorphs, graph theory

AMS(MOS) subject classifications. 05C38, 05C40, 68Q25, 68R10, 68Q10

1. Introduction. Given a graph $G = (V, E)$ and two pairs of vertices, s_1, t_1 and s_2, t_2, the two disjoint paths problem asks for vertex-disjoint paths connecting s_i with $t_i, i = 1, 2$. This problem is a special case of undirected two-commodity integral flow in which every edge has unit capacity (the problem is NP-complete if the capacities are arbitrary [GJ 78]). The problem has obvious applications in certain routing situations, and has been well studied from the point of view of sequential computation [PS 78], [Sh 80], [Se 80]. In this paper we give a fast parallel (*NC*) algorithm for it. In case G is nonplanar, our algorithm finds a Kuratowski homeomorph in G (i.e., a subgraph homeomorphic to $K_{3,3}$ or K_5). This complements the known parallel planarity algorithms, which give a planar embedding in the positive case; our algorithm provides a certificate of nonplanarity in the negative case. Our algorithms are processor efficient; in each case, the processor-time product of our algorithms is within a polylogarithmic factor of the best-known sequential algorithm.

An important step in solving the general problem is solving the planar case. A polynomial-time algorithm for this case was given by Perl and Shiloach [PS 78]. Shiloach [Sh 80] showed how to solve the nonplanar case as follows: Find a Kuratowski homeomorph. If it is a $K_{3,3}$ homeomorph, then use it as a high connectivity "switch" to find the two disjoint paths; if it is a K_5 homeomorph, use the algorithm of [Wa 68]. Independently, Seymour [Se 80] gave a polynomial-time algorithm for the decision problem for general graphs; by self-reducibility, this yields an algorithm for the search problem.

Our parallel algorithm for the planar case is based on studying properties of a new algorithmically relevant structure: the *pq-graph*. This also yields a simpler linear-time sequential algorithm. We first reduce the problem to the case where G is triconnected, and start by finding vertex disjoint paths p_1, p_2, and p_3 (q_1, q_2, and q_3) between s_1 and t_1 (s_2 and t_2). The subgraph consisting of these six paths is called a *pq-graph*. Our main structural theorem, the *pq-graph* theorem, essentially states that if this subgraph does

*Received by the editors December 1, 1988; accepted for publication (in revised form) September 16, 1991. This work was supported in part by National Science Foundation grant DCR 85-52938 and Presidential Young Investigator matching funds from AT&T Bell Laboratories and Sun Microsystems, Inc. at Cornell University. A preliminary version of this paper appeared in the Proceedings of the 30th Annual Foundations of Computer Science Conference, October 1989.

†Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742. The work of this author was partially supported by National Science Foundation grant CCR-8906949.

‡Computer Science Department, Cornell University, Ithaca, New York 14853.

§Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi 110016, India.

not contain disjoint s_i, t_i paths, $i = 1, 2$, then it must contain s_1, t_1, s_2, t_2 in this order on a single face. Using this, we prove that if the two disjoint paths do exist in G, then one of them can be taken to be one of the six paths. The sequential and parallel algorithms are now straightforward: in turn, remove each of the six paths and check whether the remaining pair of vertices is contained in a single connected component!

Our proof of the *pq-graph* theorem is patterned on the familiar "crossing lemma," which states that a cycle C with four vertices a, b, c, and d in this order cannot have a-c and b-d paths lying inside C. (This lemma follows from the Jordan Curve Theorem.) Both the crossing lemma and our *pq*-graph theorem relate the combinatorics of certain planar graphs to the geometry of their embeddings.

Our planar graph algorithm extends to $K_{3,3}$-free graphs using a theorem of Hall [Ha 43], as in [Va 89]. The main difficulty in dealing with graphs containing a $K_{3,3}$ homeomorph is finding such a homeomorph in parallel. The decision version of this problem was shown to be in *NC* in [Va 89], and parallelizing the search version was left as an open problem. We solve this problem as follows: find any Kuratowski homeomorph in the graph, and in case this is a K_5, use it to find a $K_{3,3}$ homeomorph. Once the $K_{3,3}$ homeomorph is found, we use it as a high connectivity "switch" to find the two disjoint paths; however, additional ideas are required to make this processor efficient.

We find a Kuratowski homeomorph in a nonplanar graph G as follows: use a parallel planarity testing algorithm to obtain a subgraph G' of G and an edge (u, v), such that G' is planar and $G' + (u, v)$ is nonplanar. In G', obtain a maximal cycle containing vertices u and v, and using ideas from Kuratowski's theorem find a Kuratowski homeomorph in $G' + (u, v)$.

The k-disjoint paths problem (where we need to find k vertex-disjoint paths connecting specified pairs of vertices) is NP-complete if k is part of the input [Ka 75]. For fixed $k \geq 3$, the problem was open for a long time. Recently, Robertson and Seymour have given polynomial-time algorithms for this problem for any fixed k, derived from their extensive graph minor theory [RS 86a]. The following problem is central to the Robertson–Seymour theory: for a fixed graph H, decide whether the given graph G contains H as a minor. This problem has a structure closely related to that of the k disjoint paths problem. In fact, the minor problem polynomial-time reduces (even *NC* reduces) to the k disjoint paths problem [RS 85]. Robertson and Seymour's polynomial-time algorithm for minor testing [RS 86a], together with their proof of Wagner's conjecture [RS 86b], has yielded nonconstructive polynomial-time algorithms for testing membership in any minor-closed family of graphs. For certain families (i.e., when the list of forbidden minors is known), this result gives explicit polynomial-time algorithms as well (however, because of large multiplicative factors in the running time of the minor testing algorithm, these algorithms are not practical).

A natural question is whether the Robertson–Seymour theory can be used to obtain *NC* algorithms for the k disjoint paths problem, for fixed k. Robertson and Seymour concentrate on the decision version of their problems, and rely on self-reducibility for solving the search version. We may need more structural properties in order to find parallel algorithms for the search versions. An *NC* algorithm for the k disjoint paths problem will immediately imply that testing for membership in any minor-closed family of graphs is in *NC*, without actually producing such an algorithm. As above, if the list of forbidden minors is known, this result will give an explicit *NC* algorithm as well; several natural problems fall under this category.

A dependency graph of our results is given in Fig. 1. Also listed are the time and total work bounds (processor-time product) of our parallel algorithms, as well as the

running time of the best-known sequential algorithm for each case.

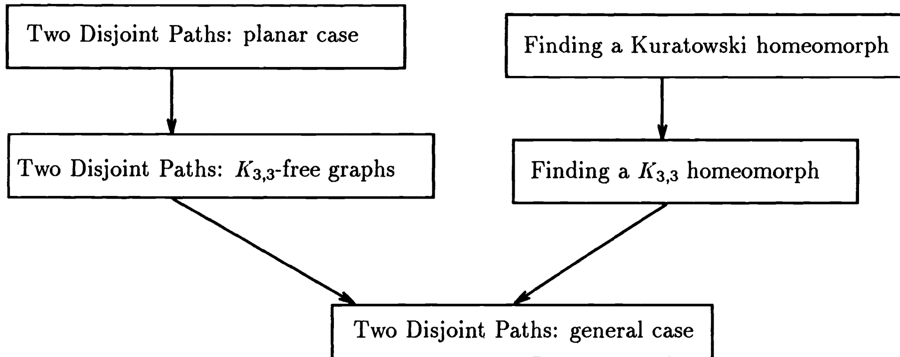


FIG. 1. *Dependency graph of results.*

The model of parallel computation used is the Concurrent-Read Concurrent-Write (CRCW) Parallel Random Access Machine (PRAM). A PRAM employs p synchronous processors, all having access to a shared memory. A CRCW PRAM allows simultaneous access by more than one processor to the same memory location for both read and write purposes. In case several processors attempt to write simultaneously in the same memory location, an arbitrary one succeeds in doing the write. (See Table 1.)

TABLE 1

Algorithm	Parallel time	Processor-time product	Sequential bound
Two paths: planar case	$O(\log n)$	$O(n \log\log n)$	$O(n)$
Kuratowski homeomorphs	$O(\log^2 n)$	$O(n \log\log n \log n)$	$O(n)$
Kuratowski homeomorphs	$O(\log n)$	$O(n^2 \log\log n)$	$O(n)$
Two paths: general case	$O(\log n)$	$O(n^2 \log\log n)$	$O(n^2)$

2. A parallel algorithm for the decision problem. We first show why we can restrict our attention to triconnected graphs.

LEMMA 2.1. *Solving the two disjoint paths* (*search or decision*) *problem for a graph G NC-reduces to solving it for the triconnected components of G.*

Proof. Decompose G into its "tree" T' of triconnected components [Va 89]. The lemma is obvious if s_i, t_i $(i = 1, 2)$ are all in the same triconnected component. Suppose s_i, t_i $(i = 1, 2)$ are in components S_i and T_i of T', respectively (see Fig. 2). Let N_i be the path from S_i to T_i in T'. If N_1 and N_2 do not intersect the disjoint paths are easy to obtain. Suppose N_1 and N_2 intersect in a path $P_0, P_1, \cdots, P_k$ in T'. Let (a_i, b_i) be the separating pair between P_i and P_{i+1}. In each triconnected component P_i find "parallel" disjoint paths $p(a_{i-1}, a_i)$, $q(b_{i-1}, b_i)$ and "cross" disjoint paths $p'(a_{i-1}, b_i)$, $q'(b_{i-1}, a_i)$. (We use the notation $p(v_i, v_j)$ for a path from vertex v_i to v_j.) If any component allows both parallel and cross paths, then using one of these we can always find the disjoint

paths. Otherwise, there is a unique way of connecting $\{a_0, b_0\}$ with $\{a_{k-1}, b_{k-1}\}$. We can now determine whether this allows disjoint s_i, t_i paths. □

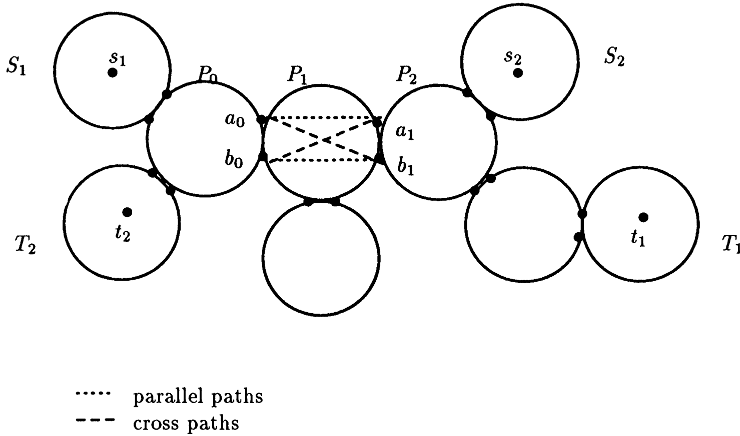


FIG. 2. *Decomposition tree T' of triconnected components.*

Henceforth we will assume that G is triconnected.

THE DECISION ALGORITHM:
Step 1. If G is nonplanar then find the separating triples in G [CKT 91], [CT91]. For each triple $\{a, b, c\}$ discard the components of $G - \{a, b, c\}$ that do not contain a vertex from $\{s_1, s_2, t_1, t_2\}$. Add virtual edges between the vertices in each triple. Let G' be the new graph. Else if G is planar let $G' = G$.
Step 2. If G' is planar then the answer to the decision version is "no" if and only if there is a face containing all four vertices s_1, s_2, t_1, t_2 in this order (by Theorem 2.2 below and the fact that a triconnected graph has a unique planar embedding [Wh 33]).
Step 3. If G' is nonplanar then the answer is "yes" [Sh 80].

THEOREM 2.2 (Perl–Shiloach). *Let G be a triconnected planar graph, and let s_1, t_1, s_2, t_2 be four vertices of G. Two disjoint paths from s_1 to t_1, and from s_2 to t_2 exist, if and only if the vertices s_1, s_2, t_1, t_2 are not on a common face in this order.*

3. Two disjoint paths: The planar case. Our algorithm for finding disjoint paths when G is planar and triconnected relies on the structure of "pq-graphs," described below. Since G is triconnected, we can find three vertex-disjoint paths p_a, p_b, and p_c from s_1 to t_1, and similarly three vertex-disjoint paths q_a, q_b, and q_c from s_2 to t_2. In [KS 91] a fast parallel algorithm is developed to solve this problem. We denote by G_{pq} the subgraph of G consisting of the p_i and q_j paths. We denote by $p[v_i; v_j]$ the segment of the path p from v_i to v_j.

DEFINITION 3.1. G_{pq} will be called a *pq-graph* if there is a numbering for the p_i and q_j paths such that when the p_i's are ordered from s_1 to t_1, and the q_j's from s_2 to t_2, the following six conditions are satisfied.

(i) The first intersection of each q_j with any p-path is a vertex of p_1.

(ii) The last intersection of each q_j with any p-path is a vertex of p_3.

(iii) Let $v_1 \in q_j \cap p_1$, $v_3 \in q_j \cap p_3$ $(v_1, v_3 \neq s_1, t_1)$. Then $q_j[v_1; v_3]$ intersects p_2.
(iv) The first intersection of each p_i with any q-path is a vertex of q_1.
(v) The last intersection of each p_i with any q-path is a vertex of q_3.
(vi) Let $v_1 \in q_1 \cap p_i$, $v_3 \in q_3 \cap p_i$ $(v_1, v_3 \neq s_2, t_2)$. Then $p_i[v_1; v_3]$ intersects q_2.

LEMMA 3.2. *If G_{pq} is not a pq-graph, then there exist disjoint s_1,t_1 and s_2,t_2 paths in G_{pq}. These paths can be obtained in $O(\log n)$ time using $O(n/\log n)$ processors.*

The proof builds on ideas from [PS 78], where they impose conditions 1–3. In fact, as can be seen from their solution, one of the paths can always be taken to be either a p path or q path. Finding the three paths and checking if the graph is a *pq-graph* can be done in parallel in the stated bounds using [CV 91], [KS 91]. (Hagerup's [Ha 90] optimal connectivity algorithm is needed for planar graphs.) We now show the following theorem.

THEOREM 3.3. *If G contains vertex-disjoint s_i, t_i paths, $i = 1, 2$, and if G_{pq} is a pq-graph, then one of these paths can be taken to be either p_2 or q_2.*

A complete proof is given in the next section.

THE SEARCH ALGORITHM FOR PLANAR GRAPHS:
Step 1. Find a planar embedding of G. If in the planar embedding s_1, s_2, t_1, and t_2 are vertices on some face F of G in this order, then there is no solution.
Step 2. Find three vertex-disjoint p-paths from s_1 to t_1 and three vertex-disjoint q-paths from s_2 to t_2. Let G_{pq} be the graph consisting of these six paths.
Step 3. Now either
(i) s_2, t_2 are in the same connected component in $G - p_i$ (for some i), or
(ii) s_1, t_1 are in the same connected component in $G - q_i$ (for some i).

Step 3 works because, if G_{pq} is not a pq-graph then, using Lemma 3.2, one of the paths can be taken to be either a p path or a q path. If G_{pq} is a pq-graph then, by Theorem 3.3, one of the two paths can be taken to be either p_2 or q_2.

THEOREM 3.4. *For planar graphs, there is an $O(\log n)$-time algorithm for the two disjoint paths search problem using $O(n \log\log n/\log n)$ processors.*

Using the parallel algorithms of [TV 85], [FRT 89], and [RR 89], we can implement the above algorithm in the stated time and processor bounds. The sequential complexity of the above algorithm matches that of [PS 78] and is $O(n)$.

Remark. The main bottleneck is the triconnectivity and planarity algorithms of [FRT 89] and [RR 89] that use $O(n \log\log n/\log n)$ processors. All the other algorithms used as subroutines are "almost" optimal [CV 91], [KS 91]. (In fact, for planar graphs one can use an optimal connectivity algorithm [Ha 90] to obtain optimal algorithms for all the other steps.) Recently, another simple sequential algorithm has been obtained by [Wo 90]. This algorithm can be parallelized by using [KS 91] as a subroutine.

4. Proof of the pq-graph theorem. The proof of Theorem 3.3 uses the Jordan regions of a cycle in a plane embedded graph to derive results about the ordering of vertices on the cycle. The crossing lemma stated below is prototypical of such an argument.

Let $G = (V, E)$ be a plane embedded graph. A cycle C partitions the plane into itself and two connected open *Jordan regions* (disjoint from C). A *face* of G is a cycle, one of whose Jordan regions is empty, that is, contains no vertices or edges of G.

Theorem 3.3 will be presented here as a corollary to the following theorem.

THEOREM 4.1. *In a pq-graph, vertices s_1, s_2, t_1, and t_2 appear in that order on a face that is vertex disjoint from p_2 and q_2 (except at the endpoints).*

COROLLARY 4.2. *If G contains vertex-disjoint s_1, t_1, and s_2, t_2 paths and if G_{pq} is a pq-graph, then one of these paths can be taken to be either p_2 or q_2.*

Proof. By Theorem 4.1 the vertices s_1, s_2, t_1, and t_2 appear in order on a face C of G_{pq}, where C is disjoint from p_2 and q_2 (except at s_1, s_2, t_1, and t_2). Define the four *sections* of C to be the subpaths from s_1 to s_2, from s_2 to t_1, from t_1 to t_2, and from t_2 to s_1. Call the empty region of C the *inside*. If G has no path inside of C between two different sections, then the vertices s_1, s_2, t_1, and t_2 lie on a face of G. (Recall that G_{pq} is a subgraph of G.)

Therefore, if the two disjoint paths exist, there is some path r in $G - G_{pq}$ inside C connecting two different sections of C. Then we can take one path to include r and edges of C, and the other can be taken to be either p_2 or q_2. □

4.1. Some geometrical lemmas. Before we prove Theorem 4.1, we present a few useful lemmas. The first is an old friend, and needs no proof.

LEMMA 4.3 (Crossing lemma). *If C is a cycle in the plane containing distinct vertices a, b, c, and d in that order, then there do not exist disjoint paths p joining a and c, and q joining b and d, both lying inside C.*

(We have already used the crossing lemma implicitly in the proof of Corollary 4.2.)

LEMMA 4.4 (Endpoint lemma). *In a pq-graph,*

(i) $s_1 \notin q_2$; (iii) $t_1 \notin q_1$; (v) $s_2 \notin p_2$; (vii) $t_2 \notin p_1$;

(ii) $s_1 \notin q_3$; (iv) $t_1 \notin q_3$; (vi) $s_2 \notin p_3$; (viii) $t_2 \notin p_2$.

Proof. Vertex s_1 cannot lie on q_2, since any path p_i must hit q_1 first before hitting q_2. The other assertions follow similarly. □

Let C be a cycle in a plane embedded graph. If vertices s and t do not lie on C, then they are *separated* by C if they lie in distinct Jordan regions of C.

Given a cycle C and a path p, a *touching* of p and C is a maximal common subpath (with one or more vertices). A *crossing* of p and C is a touching by which p crosses from one Jordan region of C to the other.

Given a path p and vertex-disjoint paths q_1, q_2, and q_3, a *segment* of p is a maximal subpath s that is disjoint from the q_i paths, except possibly at the endpoints of s. A q_i-q_j *hit* is a segment of p whose endpoints lie in q_i and q_j.

LEMMA 4.5 (Cycle lemma). *Let G be a plane embedded graph, and suppose that vertices s, t are joined by vertex-disjoint paths q_1, q_2, and q_3. Suppose that C is a cycle separating s and t. Then C has a q_1-q_3 hit.*

Remark. Symmetrically, C must also have a q_1-q_2 hit and a q_2-q_3 hit.

Proof. Since s and t lie in opposite Jordan regions of C, each path q_i must contain an odd number of C crossings. Since there are three q paths, C must contain an odd number of q path crossings.

Suppose that C has no q_1-q_3 hit. Denote by $\mathbf{Q_{13}}$ the region bounded by the cycle $q_1 q_3$, not containing q_2. Similarly define $\mathbf{Q_{12}}$ and $\mathbf{Q_{23}}$. Denote

$$\tilde{\mathbf{Q}}_{12} = \mathbf{Q_{12}} \cup q_1,$$
$$\tilde{\mathbf{Q}}_{23} = \mathbf{Q_{23}} \cup q_2.$$

Orient the cycle C. Let v be a vertex on C, and let w be the vertex preceding v in the orientation of C. Assign to v a state $Q(v)$ as follows:

$$Q(v) = \begin{cases} \tilde{\mathbf{Q}}_{\mathbf{12}} & \text{if} \quad v \in \tilde{\mathbf{Q}}_{\mathbf{12}}, \\ \tilde{\mathbf{Q}}_{\mathbf{23}} & \text{if} \quad v \in \tilde{\mathbf{Q}}_{\mathbf{23}}, \\ \mathbf{Q}_{\mathbf{13B}} & \text{if} \quad v \in q_3, \\ \mathbf{Q}_{\mathbf{13A}} & \text{if} \quad v \in \mathbf{Q}_{\mathbf{13}} \quad \text{and} \quad Q(w) = \mathbf{Q}_{\mathbf{13A}} \\ & \text{or} \quad v \in \mathbf{Q}_{\mathbf{13}} \quad \text{and} \quad Q(w) = \tilde{\mathbf{Q}}_{\mathbf{12}}, \\ \mathbf{Q}_{\mathbf{13B}} & \text{if} \quad v \in \mathbf{Q}_{\mathbf{13}} \quad \text{and} \quad Q(w) = \mathbf{Q}_{\mathbf{13B}} \\ & \text{or} \quad v \in \mathbf{Q}_{\mathbf{13}} \quad \text{and} \quad Q(w) = \tilde{\mathbf{Q}}_{\mathbf{23}}. \end{cases}$$

Then it is easy to verify that, traveling along C from an arbitrary initial vertex c_0, the possible state transitions are exactly those indicated by the diagram in Fig. 3.

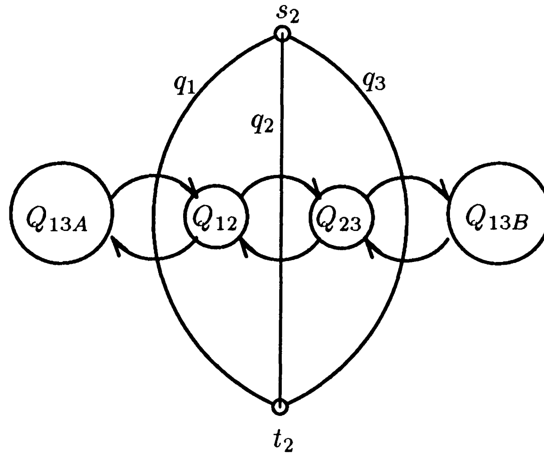


FIG. 3. *Possible state transitions for C.*

Also, it is easy to see that a cycle whose state transitions obey this diagram must have an even number of q path crossings. But we have argued that if C separates s and t, then C has an odd number of crossings. Hence C must have a q_1-q_3 hit. □

Suppose that G is a pq-graph. Define the regions $\mathbf{Q}_{\mathbf{ij}}$ and $\mathbf{P}_{\mathbf{ij}}$ as in the proof of the cycle lemma. Note that $\mathbf{Q}_{\mathbf{12}}$, $\mathbf{Q}_{\mathbf{23}}$, and $\mathbf{Q}_{\mathbf{13}}$ are pairwise disjoint, as are $\mathbf{P}_{\mathbf{12}}$, $\mathbf{P}_{\mathbf{23}}$, and $\mathbf{P}_{\mathbf{13}}$.

LEMMA 4.6 (Region lemma). *In a pq-graph*

(i) $s_1 \in \mathbf{Q}_{\mathbf{13}} \cup q_1$; (iii) $t_1 \in \mathbf{Q}_{\mathbf{13}} \cup q_3$;

(ii) $s_2 \in \mathbf{P}_{\mathbf{13}} \cup p_1$; (iv) $t_2 \in \mathbf{P}_{\mathbf{13}} \cup p_3$.

Proof. We will show that $s_2 \in \mathbf{P}_{\mathbf{13}} \cup p_1$. The other assertions follow similarly.

By the endpoint lemma, if s_2 lies on some p path, it lies on p_1. Suppose, therefore, that s_2 does not lie on any p path.

Since the first segment of q_1 is a path from s_2 to p_1, s_2 cannot lie in $\mathbf{P}_{\mathbf{23}}$. Similarly, $t_2 \notin \mathbf{P}_{\mathbf{12}}$; by the endpoint lemma, $t_2 \notin p_1 \cup p_2$. Suppose $s_2 \in \mathbf{P}_{\mathbf{12}}$. Then $p_1 p_2$ is a cycle separating s_2 and t_2. By the cycle lemma, $p_1 p_2$ must have a q_1-q_3 hit, contradicting the definition of pq-graphs. Therefore, $s_2 \in \mathbf{P}_{\mathbf{13}}$. □

Let a be a vertex on p_1. Then we say that a path q_i *straddles* a if q_i has a p_1-p_1 segment lying in $\mathbf{P_{13}}$ whose endpoints lie on p_1 on either side of a. If, furthermore, a is the first hit of some path q_j with p_1, then we say that q_i straddles s_2. We make similar definitions for s_1, t_1, and t_2.

LEMMA 4.7 (Straddle lemma). *No q path straddles s_2 or t_2. No p path straddles s_1 or t_1.*

Proof. Refer to Fig. 4. Suppose that q_i straddles a, the first hit of q_j on p_1. Let r be a segment of q_i lying in $\mathbf{P_{13}}$ with endpoints $x, y \neq a$ lying on p_1 on either side of a. Let C be the cycle formed by r and the section of p_1p_2 containing p_2. Then from the definition of a pq graph it follows that C cannot have a q_1-q_3 hit.

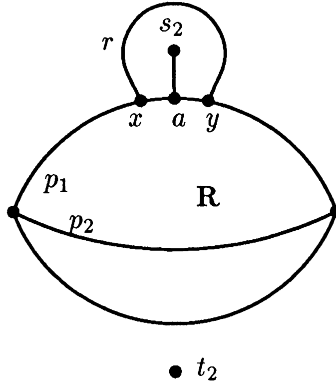


FIG. 4. *q_i cannot straddle s_2.*

Let C' be the cycle formed by r and the p_1 subpath xy. Then one region of C' is $\mathbf{R}' \subseteq \mathbf{P_{13}}$ and one region of C is $\mathbf{R} = \mathbf{R}' \cup \mathbf{P_{12}} \cup xy$. Because $a \in \mathbf{R}$ and q_j does not hit C before hitting a, $s_2 \in \mathbf{R}$.

By the region lemma, $t_2 \notin \mathbf{P_{12}}$; by the endpoint lemma, $t_2 \notin xy$. Also, $t_2 \notin \mathbf{R}'$, since the last hit of any q path is with p_3, not p_1. Therefore, $t_2 \notin \mathbf{R}$, and C separates s_2 and t_2, contrary to the cycle lemma.

Therefore, q_i cannot straddle s_2. The other assertions follow similarly. □

The proof of Theorem 4.1. Let G be a pq graph. We start by constructing a *shell* C around $G - q_2$. Eventually, we show that C is the face required by the theorem.

PROPOSITION 4.8. *$G - q_2$ has a directed cycle C (the* shell) *so that*

(i) *$G - q_2$ lies on the inside (right side) of C;*
(ii) *the outside (left side) of C lies within $\mathbf{P_{13}}$;*
(iii) *C is disjoint from p_2 (except at s_1 and t_1); and*
(iv) *C contains s_1, s_2, t_1, and t_2 in that order.*

Proof. We give the construction in three stages. (See Fig. 5.)

Stage I: Consider first the subgraph consisting of paths p_1, p_2, and p_3, embedded so that these paths appear in clockwise order around s_1. Take initially for C the cycle p_1p_3, oriented so that p_1 is directed from s_1 to t_1. Then p_2 lies on the right of C, the outside of C is $\mathbf{P_{13}}$, and C is disjoint from p_2 except for the vertices s_1 and t_1, which appear on C.

Stage II: If $s_2 \notin p_1$, then by the region lemma $s_2 \in \mathbf{P_{13}}$. In this case, add to our subgraph the segments of q_1 and q_3 from s_2 to their first hit on p_1, and reroute C to take these q segments. Do similarly if $t_2 \notin p_3$. Then the enlarged subgraph still lies inside C, the outside of C lies within $\mathbf{P_{13}}$, C is disjoint from p_2, and the vertices s_2, t_1, t_2, and s_1 appear on C in that order.

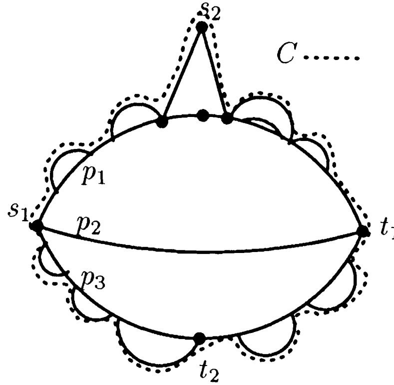


FIG. 5. *The shell C.*

Stage III: Since every q path now begins and ends on C, any q subpath that escapes C must return to C. Add q_1 to our subgraph segment by segment, being careful to reroute C along every segment that would otherwise escape. Let r be such a segment. The endpoints of r must lie on a p path. Since r cannot have a p_1-p_3 hit, r does not straddle s_1 or t_1. By the straddle lemma, r does not straddle s_2 or t_2. Therefore, C still contains s_2, t_1, t_2, and s_1 in that order. Also, C continues to avoid p_2 and enclose the entire subgraph. In a similar manner, add q_3. □

The remainder of the proof is devoted to showing that q_2 also lies inside C. The next two propositions show that C is well behaved.

Denote by C^+ the oriented subpath of C from s_1 to s_2 to t_1. Similarly define C^- from t_1 to t_2 to s_1.

PROPOSITION 4.9. (i) *C^+ contains the first hit of p_1 with q_1;*
(ii) *C^+ contains the last hit of p_1 with q_3;*
(iii) *C^- contains the first hit of p_3 with q_1;*
(iv) *C^- contains the last hit of p_3 with q_3;*
(v) *C^+ contains the first hit of q_1 with p_1;*
(vi) *C^+ contains the first hit of q_3 with p_1;*
(vii) *C^- contains the last hit of q_1 with p_3;*
(viii) *C^- contains the last hit of q_3 with p_3;*

Proof. (i) If $s_1 \in q_1$ then we need only observe that $s_1 \in C^+$. Otherwise, observe that the first hit c of p_1 with q_1 lies on C^+ in Stage I of the above construction, and cannot be straddled by any q_1 or q_3 segment in Stages II or III. Statements (ii)–(iv) follow similarly.

(v) If $s_2 \in p_1$ then we need only observe that $s_2 \in C^+$. Otherwise, observe that the first hit d of q_1 with p_1 lies on C^+ in Stage II of the above construction, and cannot be straddled by any q_1 or q_3 segment in Stage III. Statements (vi)–(viii) follow similarly. □

PROPOSITION 4.10. *Suppose $s_2 \notin p_1$, and denote by a and b the first vertices of q_1 and q_3 upon p_1. Then these vertices appear on C^+ in the order s_1, a, s_2, b, t_1. Analogous statements hold at t_1, t_2, and s_1.*

Proof. Let c denote the first hit of p_1 on q_1, and let d denote the last hit of p_1 on q_3. By the previous proposition a, b, c, and d lie on C^+, with c being the first, and d being the last. By the crossing lemma applied to C and c, b, a, and d, b cannot precede a. By another application of the crossing lemma to c, b, s_2, and d, b cannot precede s_2. Similarly, s_2 cannot precede a. □

We now consider a cycle inside C, made up mostly of q_1 and q_3, which we will show contains q_2 on its inside.

Let g be the path inside C from s_1 to t_1 constructed by following p_1 from s_1 to q_1, q_1 to s_2, q_3 from s_2 to the last hit of q_3 along p_1, and p_1 to t_1. Similarly, define g' from t_1 to t_2 to s_1. Then by the preceding proposition and an application of the crossing lemma g and g' are vertex-disjoint (except for s_1 and t_1). We will show that q_2 lies inside gg'. First we prove the following.

PROPOSITION 4.11. *Consecutive hits of g with C^+ appear in forward order along C^+. Similarly, consecutive hits of g' with C^- appear in forward order along C^-.*

Proof. We show that g cannot pass through a vertex a on C^+ and subsequently hit a vertex b, which precedes a on C^+. We proceed by induction on a along C^+. Initially, we take $a = s_1$. (See Fig. 6.)

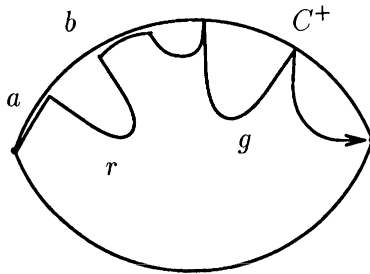


FIG. 6. *Consecutive g hits appear in order along C^+.*

If $a \neq t_1$ then g must proceed to hit C^+ at some vertex b, which, by the induction hypothesis, succeeds a along C^+. If $b = t_1$, then g does not proceed beyond b. Suppose that $b \neq t_1$, and let r denote the a-b subpath of g. Since g does not self-intersect, g will hit neither a nor b again. Suppose that g hits some vertex $c \in C^+$ lying strictly between a and b. Let r' denote the subpath of g from c to t_1. Then r and r' violate the crossing lemma applied to C. Therefore, g subsequently never hits any vertex preceding b on C^+. □

So far we have shown that the shell C is a face of the graph $G - q_2$ on which the vertices s_2, t_1, t_2, and s_1 appear in the given order, and that C is disjoint from p_2 (except for s_1 and t_1). We note that q_2 is disjoint from the cycle gg' (except at s_2 and t_2). All that remains to be shown is that q_2 lies inside the cycle gg', implying that q_2 (and hence all of G) lies inside C, and that q_2 is vertex-disjoint from C (except at s_2 and t_2).

We shall need the following. We say that gg' has four sections, meaning the t_2-s_1 subpath (section I), the s_1-s_2 subpath (section II), the s_2-t_1 subpath (section III), and the t_1-t_2 subpath (section IV).

PROPOSITION 4.12. *No gg' segment of p_2 lying outside (to the left of) gg' has endpoints in two different sections.*

Proof. Suppose that p_2 has a gg' segment xy lying to the left of gg', with x and y in different sections of gg'. By the endpoint lemma $s_2, t_2 \notin p_2$, so neither x nor y is equal to s_2 or t_2. Therefore, both gg' paths from x to y contain an s or t vertex, and in particular both paths hit C at some point strictly between x and y. If $x \neq s_1$ or t_1, let a be the last vertex of gg' (strictly) before x which lies on C, and let b be the next vertex of gg' (strictly) after x which lies on C. (See Fig. 7.) If $x = s_1$ or t_1, the choice of a and b depends on the direction of the p_2 segment. gg' hits y strictly after hitting b (and before returning again to a), so the cycle formed by the ab subpath of gg' and C encloses y in

the region to its left. But gg' hits C after hitting y, so that gg' hits C after hitting b at a vertex between a and b. This contradicts the previous proposition. □

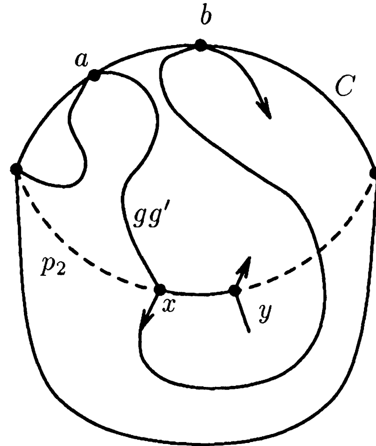


FIG. 7. *gg' cannot reach y.*

PROPOSITION 4.13. *Path q_2 lies inside (to the right of) gg'.*

Proof. Partition p_2 into its gg' segments. Since s_1 lies in sections I and II, and t_1 lies in sections III and IV, some gg' segment r of p_2 crosses from either section I or II, to either III or IV. By the previous proposition, r lies inside gg'. Segment r must hit q_1 first, q_3 last, and in between r must hit an internal vertex of q_2. Since q_2 is disjoint from gg', all of q_2 lies inside gg'. □

This concludes the proof of Theorem 4.1

5. Extracting a Kuratowski homeomorph.

DEFINITIONS. Let C be a cycle in G, and let e and f be edges of G not in C. Define the equivalence relation $=_C$ by $e =_C f$ if and only if there is a path in G that includes e and f and has no internal vertices in common with C. The subgraphs induced by the edges of the equivalence classes of $E(G) - E(C)$ under $=_C$ are called the *bridges* of G relative to C. The *vertices of attachment* of bridge B to cycle C are the vertices in $V(B) \cap V(C)$.

A bridge with k vertices of attachment is called a k-bridge. Two k-bridges with the same vertices of attachment are *equivalent* k-bridges. The vertices of attachment of a k-bridge B with $k \geq 2$ effect a partition of C into edge-disjoint paths, called the *segments* of B. Two bridges *avoid* one another if all the vertices of attachment of one bridge lie in a single segment of the other bridge; otherwise they *overlap*. Two bridges B and B' are *skew* if there are four distinct vertices u, v, u', v' of C such that u and v are vertices of attachment of B, u' and v' are vertices of attachment of B', and the four vertices appear in the order u, u', v, v' on C. It is shown in [BM 77] that if two bridges overlap, then they are either skew or equivalent three-bridges.

If C is a Jordan curve in the plane, then the rest of the plane is partitioned into two disjoint open sets called the *interior* and *exterior* of C. We denote the closures of the regions by Int$C$ and Ext$C$, respectively. In a plane graph G, each bridge of G relative to C is entirely contained in Int$C$ or Ext$C$. A bridge in Int$C$ (Ext$C$) is called an *inner* (*outer*) bridge.

LEMMA 5.1. *If G is planar, and $G + e$ is nonplanar and triconnected, then we can find a Kuratowski homeomorph in $G + e$ in $O(\log n)$ time with $O(n \log\log n / \log n)$ processors.*

Proof. Let u and v be the endpoints of e. The proof of Kuratowski's theorem in [BM 77] relies on finding a cycle C of G that contains u and v and is such that the set of edges in Int$C$ is maximal. We give a parallel algorithm for finding such a *maximal cycle*. The rest of the proof follows by a case analysis given in [BM 77]. Since G is biconnected, we can find a cycle C in G containing u and v by finding two disjoint u-v paths (see [KS 91]). Now consider the set of bridges of G with respect to the cycle C. Given a planar embedding of G, the bridges may be partitioned into two sets:

$$\text{Outer}_C = \{B_i \mid B_i \text{is embedded in Ext}C\},$$

$$\text{Inner}_C = \{B_j \mid B_j \text{is embedded in Int}C\}.$$

A bridge $B_i \in \text{Outer}_C$ is in the set Outer-skew$_C$ if B_i is skew to (u, v); B_i is in the set Outer-nonskew$_C$ if it is not skew to (u, v). Since $G + e$ is nonplanar, there is at least one bridge in the set Outer-skew$_C$ (otherwise e can be embedded in Ext$C$ in a planar embedding of G).

We first modify the cycle C to obtain a cycle C^1 containing u and v so that there are no bridges in the set Outer-nonskew$_{C^1}$ with respect to the cycle C^1. We then show how to modify C^1 to obtain cycle C^3 containing u and v such that the bridges in set Outer-skew$_{C^3}$ are single edges skew to (u, v).

Since G is biconnected, each bridge has at least two vertices of attachment on C. Let the attachment vertices of B_i on C be $x_i^1, x_i^2, \cdots, x_i^{k_i}$ (considered in clockwise order on C). We call $x_i^1 (x_i^{k_i})$ the first (last) attachment vertex of B_i on C. The segment $C[x_i^1; x_i^{k_i}]$ is called the *attachment bar* of B_i on C. It was shown in [BM 77] that outer bridges avoid one another. Hence, the attachment bar of each outer bridge can overlap with the attachment bar of another outer bridge only at an end vertex, and not at any internal vertex. Each bridge in the set Outer-nonskew$_C$ has all of its attachment vertices (and thus its attachment bar) on the segment $C[v; u]$ or $C[u; v]$.

Consider the planar embedding of B_i and its attachment bar, and call the resultant graph B_i' (see Fig. 8). Note that B_i' is biconnected and planar. In B_i' the vertices x_i^1 and $x_i^{k_i}$ are on the outermost face in the planar embedding. Since B_i' is biconnected, the outer face of B_i' is a simple cycle C_i'. Let $P_i(x_i^1, x_i^k)$ be the subpath of C_i' from x_i^1 to $x_i^{k_i}$ avoiding the the attachment bar of B_i.

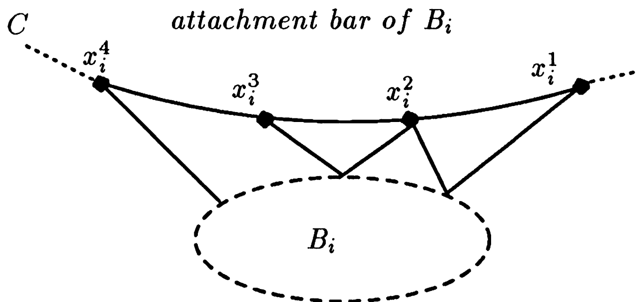


FIG. 8. *The graph* B_i'.

In C, replace all the segments that are attachment bars of some bridge $B_i \in$ Outer-nonskew$_C$ by the path $P_i[x_i^1; x_i^{k_i}]$. The new cycle C^1 contains both u and v, and all its outer bridges are skew to (u, v).

We perform the transformation from C^1 to C^3 in two stages. Consider the cycle C^1 and its bridges B_i in ExtC^1, which are skew to (u, v). If B_i has only two attachment

vertices, then clearly the bridge is only a path, and in fact, since the graph is triconnected, it is only a single edge. Assume that B_i has more than two attachment vertices on C^1. Let the attachment vertices on $C^1[u;v]$ be $x_i^1, x_i^2, \cdots, x_i^{k_i}$ (considered in order). The attachment vertices on $C^1[v;u]$ are $y_i^1, \cdots, y_i^{l_i}$ (see Fig. 9). We will replace $C^1[x_i^1; x_i^{k_i}]$ by a path P' in B_i to obtain C^2; and subsequently replace $C^2[y_i^1; y_i^{l_i}]$ by a path P'' in B_i to obtain C^3, the maximal cycle.

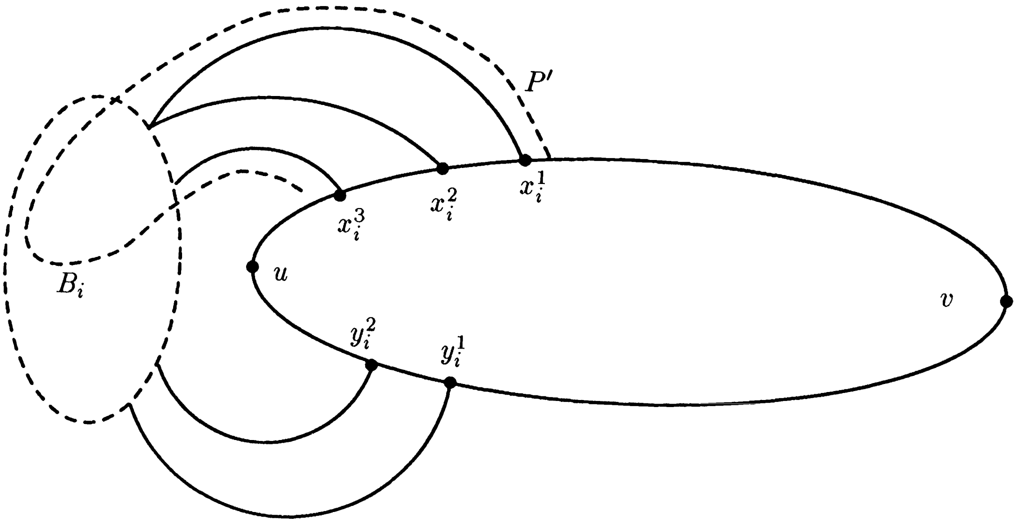


FIG. 9. *Bridge with attachment vertices skew to (u, v).*

Obtain the subgraph B_i^1 of B_i by deleting the attachment vertices y_i on $C^1[v;u]$ (see Fig. 10). Consider the external face in a planar embedding of B_i^1. Find a path P from x_i^1 to $x_i^{k_i}$ along the external face. This path may not be simple, as shown in Fig. 11. Find the articulation vertex closest to x_i^1 on P, and obtain a simple path P' by "short-circuiting" P. Replace the segment $C^1[x_i^1; x_i^{k_i}]$ by the path P' to obtain the cycle C^2. Do this step in parallel for every bridge in ExtC^1 skew to (u, v). Now identify the subgraphs B_i of G, which form bridges with respect to the new cycle C^2. Let x_i^j and y_i^j be the attachment vertices of B_i on the upper and lower chains $C^2[u;v]$ and $C^2[v;u]$. Obtain the subgraph B_i^2 of B_i by deleting the attachment vertices x_i^j of B_i on $C^2[u;v]$. In a planar embedding of B_i^2 replace the segment $C^2[y_i^1; y_i^{l_i}]$ by the path P'' from y_i^1 to $y_i^{l_i}$ in the external face. If G is triconnected it is easy to see that the path P'' is simple (see Fig. 12). After the replacement we obtain cycle C^3; the only bridges in ExtC^3 are single edges skew to (u, v).

Consider the set Inner$_{C^3}$ of inner bridges of C^3. Since $G + e$ is nonplanar, the set Inner-skew$_{C^3}$ is nonempty (else a planar embedding for $G + e$ can be obtained). If each bridge in Inner-skew$_{C^3}$ avoids every bridge in Outer$_{C^3}$, then each such bridge can be transferred to ExtC^3, yielding a planar embedding for $G + e$. Hence, there must be an inner bridge B_1 that overlaps an outer bridge B_2 and that is skew to (u, v).

At this point, by considering various cases for the possible configurations of attachment vertices of B_1 and B_2 on C^3, we can obtain a Kuratowski homeomorph as in [BM 77]. The parallel algorithm takes $O(\log n)$ time using $O(n \log\log n / \log n)$ processors using [RR 89], [CV 91], [KS 91]. □

From the proof of Lemma 5.1, we also conclude the following.

LEMMA 5.2. *In a planar graph G, we can a find a maximal cycle C that contains u and*

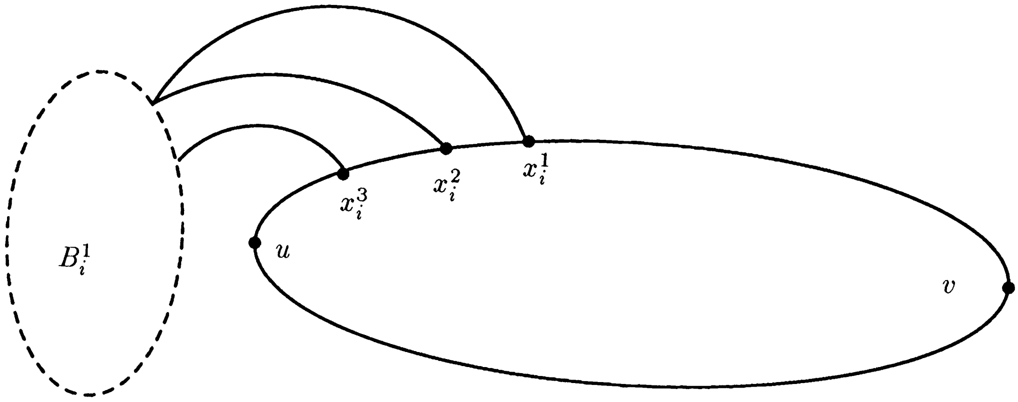


FIG. 10. *Subgraph* B_i^1.

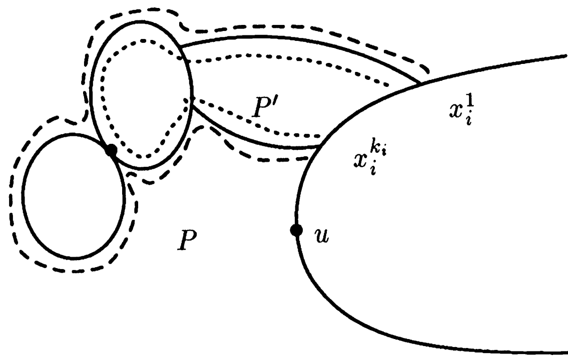


FIG. 11. *General form of path* P'.

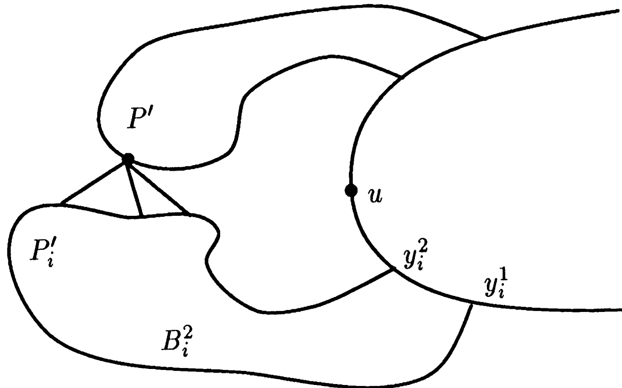


FIG. 12. *Path* P_i' *is simple.*

v in $O(\log n)$ time using $O(n \log\log n/ \log n)$ processors.

Lemma 5.1 yields the parallel algorithm described below.

ALGORITHM FOR FINDING A KURATOWSKI HOMEOMORPH:
Step 1. Order the edges in E arbitrarily as $e_1, e_2, \cdots, e_m$. Let G_i be the subgraph consisting of the first i edges. Clearly, G_1 is planar, and G_m is not.
Step 2. Find the smallest index k such that $G_1, G_2, \cdots, G_{k-1}$ are all planar, but G_k $(= G_{k-1} + e_k)$ is not, using the planarity testing algorithm of [RR 89].
Step 3. Find a triconnected nonplanar component of G_k; call it G'_k. Note that $e_k \in G'_k$ and that $G'_k - e_k$ is planar and biconnected. Apply Lemma 5.1 to the graph $G'_k - e_k$ and edge e_k to find a Kuratowski homeomorph.

Remarks. We can assume that G has $O(n)$ edges; if not, we can select any $3n - 5$ of them, keeping G nonplanar. Step 2 can be implemented either by checking all the graphs G_i in parallel for planarity, or by doing a binary search on the graphs G_i. Using the algorithm of [RR 89], that takes $O(\log n)$ time using $O(n \log\log n/ \log n)$ processors, we can implement the above algorithm to run in $O(\log n)$ time with $O(n^2 \log\log n/ \log n)$ processors, or in $O(\log^2 n)$ time using $O(n \log\log n/ \log n)$ processors. We leave open the problem of finding a Kuratowski homeomorph in the same time and processor bounds as for planarity testing, i.e., $O(n \log\log n/ \log n)$ processors and $O(\log n)$ time.

THEOREM 5.3. *In a nonplanar graph G, a Kuratowski homeomorph can be obtained in $O(\log n)$ time using $O(n^2 \log\log n/ \log n)$ processors, or it can be obtained in $O(\log^2 n)$ time using $O(n \log\log n/ \log n)$ processors.*

6. Finding a $K_{3,3}$ homeomorph. Our algorithm for finding a $K_{3,3}$ homeomorph is a parallelization of the sequential algorithm given in [As 85]. This algorithm is based on the following theorem of Hall [Ha 43].

THEOREM 6.1 (Hall). *Each triconnected component of a $K_{3,3}$-free graph (i.e., a graph not containing a $K_{3,3}$ homeomorph) is either planar or exactly the graph K_5.*

THEOREM 6.2. *There is an $O(\log n)$ time algorithm using $O(n^2 \log\log n/ \log n)$ processors, or an $O(\log^2 n)$ algorithm using $O(n \log\log n/ \log n)$ processors, which finds a $K_{3,3}$ homeomorph (if one exists).*

Proof. First decompose G into its triconnected components. By Hall's theorem, one of the components is nonplanar and contains at least six vertices. Use the algorithm of Theorem 5.3 to find any Kuratowski homeomorph G'. In case it is a homeomorph of K_5, call the vertices in G' of degree four v_1, v_2, v_3, v_4, and v_5. If G' is exactly the graph K_5, then the triconnected component containing G' must contain another vertex v not in G'. Obtain three vertex-disjoint paths from v to any three vertices of the K_5, say v_1, v_2, and v_3. These can be found by introducing an artificial sink vertex u and three edges from u to v_1, v_2, and v_3, and then finding three vertex-disjoint v-u paths. From the three paths and the K_5 it is easy to find a subgraph homeomorphic to $K_{3,3}$ by putting v_1, v_2, and v_3 in one partition of the $K_{3,3}$ and v, v_4, and v_5 in the other.

If the homeomorph G' is a subdivision of K_5, then consider a vertex u on the path $P(v_1, v_2)$, where v_1 and v_2 are vertices of degree four in G'. (P is a subdivision of an edge of the K_5.) Since G is triconnected, there must be a path in $G - \{v_1, v_2\}$ from u to some other vertex w of G'. Use this path to extract a $K_{3,3}$ homeomorph by considering where the path first hits G' (details in [As 85]). □

7. Two disjoint paths: The nonplanar case. We now develop the algorithm to solve the two paths problem for the case of general graphs. When the graph is planar, we can use the algorithm outlined earlier. If the graph is a $K_{3,3}$-free graph, then using a theorem

by Hall [Ha 43] it is easy to solve the problem by reducing it to the planar graph case. Henceforth, we will assume that the graphs are nonplanar and contain homeomorphs of $K_{3,3}$.

The algorithm for the case of general graphs proceeds by first finding a Kuratowski homeomorph in a nonplanar graph. In case the obtained homeomorph is of a K_5, we convert it to a $K_{3,3}$. Using the $K_{3,3}$ it becomes easy to obtain a parallel algorithm using the approach by [Sh 80].

Suppose G contains a subgraph $G_{3,3}$ homeomorphic to $K_{3,3}$. Call the nine paths of $G_{3,3}$ representing the edges of $K_{3,3}$ *p-edges*, and call the six vertices of $G_{3,3}$ representing the six vertices of $K_{3,3}$ *p-vertices*.

ALGORITHM FOR TRICONNECTED GRAPHS HAVING A $K_{3,3}$ HOMEOMORPH:
Step 1. Find the separating triples in G [CKT 91]. For each triple $\{a, b, c\}$ discard the components of $G - \{a, b, c\}$ which do not contain a vertex from $\{s_1, s_2, t_1, t_2\}$. Add virtual edges between the vertices in each triple. Let G' be the new graph. (This is needed for Step 5; for details see [Sh 80].)
Step 2. Find a $K_{3,3}$ homeomorph in G' (else G' is $K_{3,3}$-free); call it $G_{3,3}$.
Step 3. Modify $G_{3,3}$ to include s_1 as one of the "corner" vertices. This is done by finding three-disjoint paths from s_1 to $G_{3,3}$. Call the modified homeomorph $G'_{3,3}$.
Step 4. Modify $G'_{3,3}$ further to include t_1 either as a corner vertex or on a p-edge incident to s_1. Call this homeomorph $G''_{3,3}$.
Step 5. Find four vertex-disjoint paths π_1, π_2, π_3, and π_4 connecting s_1, t_1, s_2, and t_2 with four p-vertices of $G''_{3,3}$ different from s_1 and t_1. These paths together with the $K_{3,3}$ homeomorph yield the two disjoint s_i, t_i, $i = 1, 2$, paths.

Step 1 can be parallelized using the algorithm in [CKT 91]. Theorem 6.2 yields a parallel algorithm for step 2. For steps 3 and 4, Shiloach gives a sequential path extension algorithm which modifies $G_{3,3}$. We parallelize this by working with *prefixes*. We describe the procedure in detail for step 3; the idea for step 4 is similar. Our case analysis is made more complicated by the fact that we *do not* do the preprocessing done by Shiloach, to obtain the "W-assumption" about the nature of the paths.

We make s_1 a p-vertex by constructing three-disjoint paths P_1, P_2, and P_3 from s_1 to three p-vertices on the same side of $G_{3,3}$. Let P_i have $v_i^1, v_i^2, \cdots, v_i^{k_i}$ as its vertices of intersection with the p-edges (considered in order from s_1). Let $P_i^1, \cdots, P_i^{k_i}$ denote the segments $P_i[s_1; v_i^1], \cdots, P_i[s_1; v_i^{k_i}]$, respectively. We define the *set of prefixes* (i, j, k) *of* P as $\{P_1^i, P_2^j, P_3^k\}$.

LEMMA 7.1. *The subgraph $G_{3,3}$ and some set of prefixes yields a subgraph $G'_{3,3}$ of G homeomorphic to $K_{3,3}$, with s_1 as a p-vertex.*

Proof. The proof is based on an exhaustive case analysis. First consider the set $(1, 1, 1)$ of P. In case the prefixes are incident on different p-edges or on p-vertices, use one of the base cases from [Sh 80], for which no extension of any prefix is required to get $G'_{3,3}$. The nontrivial case is that in which all three prefixes are incident on the same p-edge, say e_1, with endpoints v_1 and v_2. In this case extend the middle prefix to its next intersection with a p-edge (or a p-vertex). There are several cases to consider:

1. The extended segment hits e_1.
2. The extended segment hits a p-edge different from e_1.
3. The extended segment hits a p-vertex different from v_1 and v_2.

In the first case, continue extending the segments by extending the middle prefix at each step. Since the P_i-paths end at different p-vertices, this construction eventually falls into case 2 or 3. By extending only the middle prefix, we ensure that the middle prefix

does not intersect the two end segments $[v_1; x_1]$ and $[v_2; x_2]$ of e_1, where x_1 and x_2 are the end vertices of the nonmiddle prefixes. With this condition, in the last two cases the solution follows from Shiloach's base cases.

We illustrate this construction with an example. In Fig. 13 the middle prefix is of path P_2, which we extend to its second hit with a p-edge. If it hits a p-edge other than e_1, we immediately apply case 2. Assume that it is e_1. Now the prefix of P_3 becomes the middle prefix, which we extend to its second hit with a p-edge. In this case it is a p-edge different from e_1; apply a base case to obtain $G'_{3,3}$ (see Fig. 14).

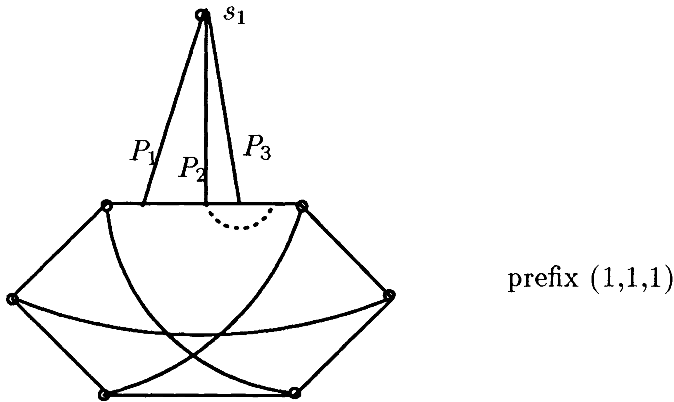


FIG. 13. *Extend the middle prefix P_2.*

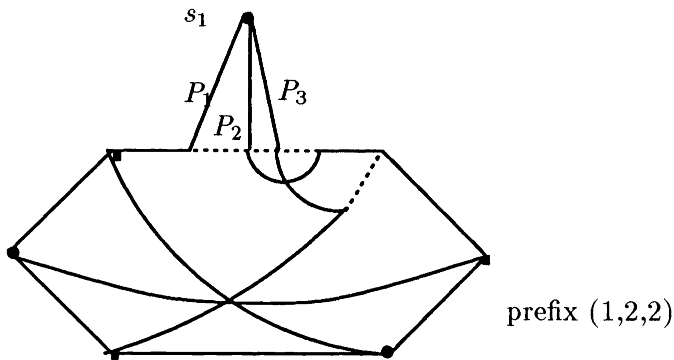


FIG. 14. *Extending P_3 yields a base case.*

We illustrate a sequence of extensions of the middle prefixes in Fig. 15; the sequence of extensions terminates when P_2 hits a p-edge different from e_1. □

We now develop a parallel algorithm for finding the subgraph $G'_{3,3}$. First check all the base cases (when the first hits of the P_i paths yield $G'_{3,3}$). The only nontrivial case is when the first hits of all three P_i paths are on p-edge e_1. Now "guess" the ending configuration (there are only six of them) of the prefixes of the paths P_1, P_2, and P_3 on e_1. Extend the middle path (say P_2) of the guessed configuration until it first hits a p-vertex or an edge different from e_1. This can be done in $O(\log n)$ time using $O(n/\log n)$ processors. After extending P_2, compute the leftmost and rightmost hits (x_l and x_r) of P_2 with p-edge e_1. The segments $e_1[v_1; x_l]$ and $e_1[x_r; v_2]$ are free of intersection with P_2. Extend

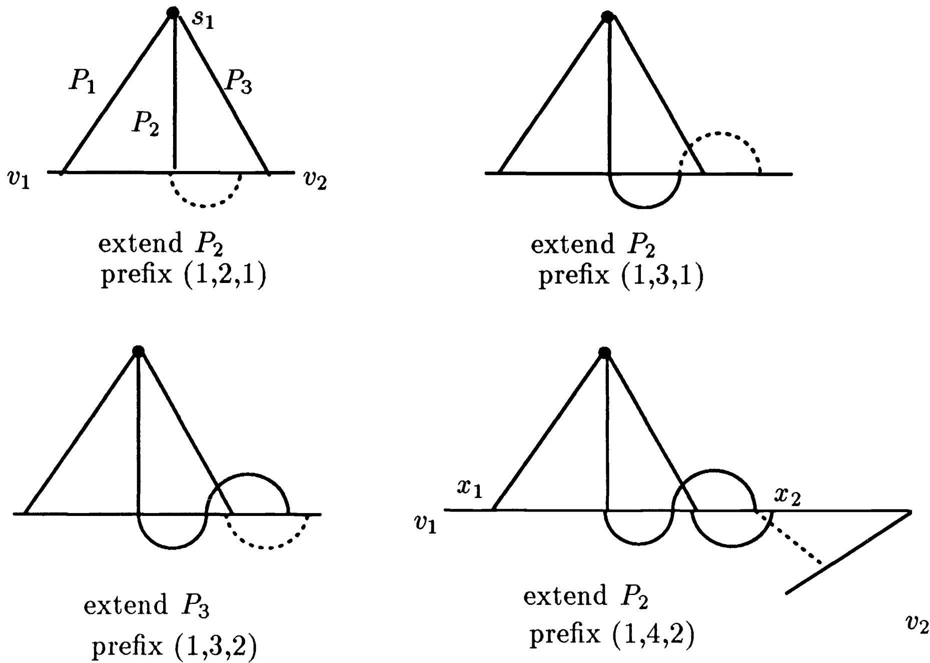


FIG. 15. *A sequence of extensions of prefixes of the P-paths.*

paths P_1 and P_3 independently until they land on these segments in the same order as in the guessed configuration. We check to ensure that they do not have intersections with any p-edge other than e_1. (If they do, then we guessed the configuration incorrectly.)

Shiloach showed further that we can obtain $G''_{3,3}$ with s_1 as a p-vertex and with either

1. t_1 also as a p-vertex, or
2. t_1 lying on a p-edge incident with s_1.

The construction is similar to that given above. Find disjoint paths Q_1, Q_2, and Q_3 connecting t_1 with three distinct p-vertices of $G'_{3,3}$; now there are more cases to consider, since the prefixes of the Q_i paths may land on p-edges incident to s_1 or on p-edges not incident to s_1.

LEMMA 7.2. *The subgraph $G'_{3,3}$ and some set of prefixes yields $G''_{3,3}$ satisfying conditions* 1 *and* 2.

This construction is similar to the above construction and can be carried out in parallel.

For Step 5 we introduce artificial source and sink vertices to construct the four π_i-paths connecting s_1, t_1, s_2 and t_2 with four p-vertices on $G''_{3,3}$ that are different from s_1 and t_1 (these are guaranteed to exist by Step 1). Shiloach showed how $G''_{3,3}$ could be used to make two disjoint connections, one between π_1 and π_2 and the other between π_3 and π_4, to yield the desired disjoint s_i, t_i paths.

There are three cases to consider:

1. s_1 and t_1 are p-vertices not connected by a p-edge.
2. s_1 and t_1 are p-vertices connected by a p-edge.
3. t_1 is on a p-edge incident with s_1.

Shiloach showed that by considering only the first hits of π_3 and π_4 we can obtain a solution for case 1.

The solution for case 2 is obtained as follows: the only nontrivial subcase (the others are symmetric) is as shown in Fig. 16. We first describe an $O(n)$ sequential "game" played by paths π_1, π_3, and π_4 on p-edges e_2 and e_3. (Our game is a modification of Shiloach's sequential algorithm.) We say that a prefix of path π_i *covers* a prefix of π_j if they land on the same p-edge with the endpoint of π_j closer to s_1 than the endpoint of π_i. The game starts with π_3. Inductively, we extend the current π path until it lands uncovered on p-edge e_2 or e_3, or until it leaves the game altogether by landing on a p-edge different from e_1, e_2, or e_3. Ultimately, one of the three paths leaves the game and we obtain prefixes allowing disjoint π_1-π_2 and π_3-π_4 connections. See Figs. 17 and 18.

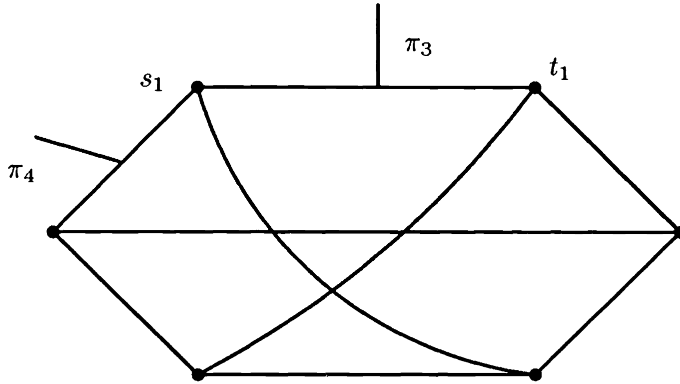


FIG. 16. *The only nontrivial subcase for case* 2.

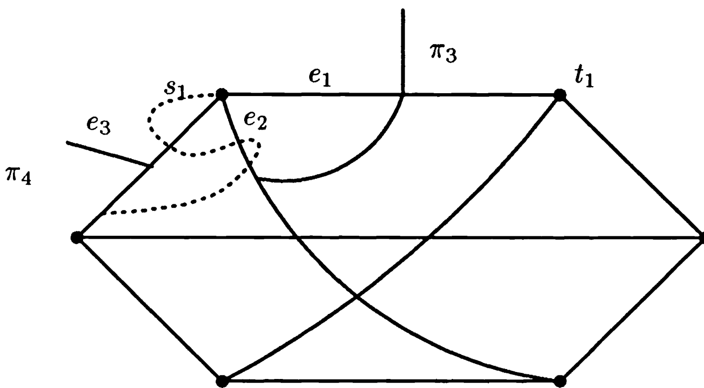


FIG. 17. *Paths π_3 and π_4 cover π_1.*

We give a parallel algorithm to obtain such a prefix. First guess which p-edge each path ends on, and which path has left the game. There are only six distinct configurations and we check all of them. Suppose we guess that the first path to leave is π_1. Extend π_1 to its first hit on some edge other than e_1, e_2, or e_3. (The appropriate prefix of π_1 can be obtained by a prefix computation in $O(\log n)$ time using $O(n/\log n)$ processors.) Extend π_3 and π_4 to their shortest prefixes on edges e_2 and e_3 that are not covered by the extended prefix of π_1. If our guess was correct, then from the success of the sequential game we know that the extensions of π_3 and π_4 hit no p-edges besides e_1, e_2, and e_3. It is now easy to obtain the required disjoint connections.

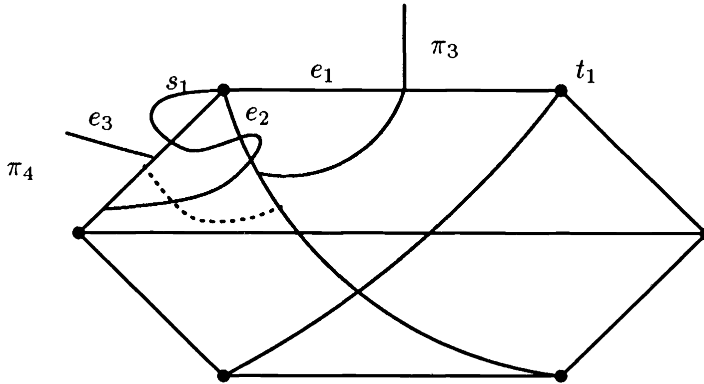


FIG. 18. *Path π_1 covers π_4.*

The solution for case 3 is very similar to the solution for case 2. The only difference is that all four π_i paths participate in the game on all three edges e_i. Ultimately, one of the four paths leaves the game to hit another edge. We guess which path is the first to leave and extend the other paths until they are not covered by the extended prefix of the first one. A solution can now be obtained (see [Sh 80] for details).

LEMMA 7.3. *One of the sets of prefixes of π_i along with the modified $G_{3,3}$ provides the disjoint π_1-π_2 and π_3-π_4 connections.*

Remark. Our Lemmas 7.1–7.3 avoid Shiloach's "W-assumption," which relies on a sequential process at the expense of more complicated case analyses.

The bottlenecks in the algorithm are Step 1 [CKT 91] and finding a $K_{3,3}$ homeomorph (for which it is better to use the inefficient algorithm since it gives a better parallel time bound).

THEOREM 7.4. *Given a graph G and vertices s_1, t_1, s_2, and t_2, we can solve the two paths problem in $O(\log n)$ time using $O(n^2 \log\log n/\log n)$ processors.*

Acknowledgments. Samir Khuller thanks the people at the IBM T. J. Watson Research Center for the pleasant environment provided during the summer of 1988.

REFERENCES

[As 85] T. ASANO, *An approach to the subgraph homeomorphism problem,* Theoret. Comput. Sci., 38 (1985), pp. 249–267.

[BM 77] J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, American Elsevier, New York, 1977.

[CKT 91] J. CHERIYAN, M. KAO, AND R. THURIMELLA, *Scan-first search and sparse certificates: An improved parallel algorithm for k-vertex connectivity,* Tech. Report CS-1991-20, Dept. of Computer Science, Duke University, Chapel Hill, NC.

[CT 91] J. CHERIYAN AND R. THURIMELLA, *Algorithms for parallel k-vertex connectivity and sparse certificates,* Proc. 23rd Annual Symposium on Theory of Computing, May 1991, pp. 391–401.

[CV 91] R. COLE AND U. VISHKIN, *Approximate parallel scheduling* II. *Applications to logarithmic time optimal parallel graph algorithms,* in Inform. Comput., 92 (1991), pp. 1–47.

[FRT 89] D. FUSSELL, V. RAMACHANDRAN, AND R. THURIMELLA, *Finding triconnected components by local replacements*, in Proc. 16th Internat. Conference on Automata, Languages, and Programming, Lecture Notes in Comput. Sci. 372, Springer-Verlag, New York, 1989, pp. 379–393; SIAM J. Comput., to appear.

[GJ 78] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractibility: A Guide to the Theory of NP-Completeness,* Freeman, San Francisco, 1978.

[Ha 90] T. HAGERUP, *Optimal parallel algorithms on planar graphs*, Inform. Comput., 84 (1990), pp. 71–96.

[Ha 43] D. W. HALL, *A note on primitive skew curves*, Bull. Amer. Math. Soc., 49 (1943), pp. 935–937.

[Ka 75] R. M. KARP, *On the complexity of combinatorial problems*, Networks, 5 (1975), pp. 45–68.

[KS 91] S. KHULLER AND B. SCHIEBER, *Efficient parallel algorithms for testing k-connectivity and finding disjoint s-t paths in graphs*, SIAM J. Comput., 20 (1991), pp. 352–375.

[PS 78] Y. PERL AND Y. SHILOACH, *Finding two disjoint paths between two pairs of vertices in a graph*, J. Assoc. Comput. Mach., 25 (1978), pp. 1–9.

[RR 89] V. RAMACHANDRAN AND J. H. REIF, *An optimal parallel algorithm for graph planarity*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, October 1989, pp. 282–287. (See also *Planarity testing in parallel*, TR-90-15, Dept. of Computer Science, University of Texas at Austin.)

[RS 85] N. ROBERTSON AND P. D. SEYMOUR, *Disjoint paths—A survey*, SIAM J. Algebraic Discrete Methods, 6 (1985), pp. 300–305.

[RS 86a] ———, *Graph Minors* XIII: *The disjoint paths problem*, manuscript, 1986.

[RS 86b] ———, *Graph Minors* XVI: *Wagner's Conjecture*, manuscript, 1986.

[Se 80] P. D. SEYMOUR, *Disjoint paths in graphs*, Discrete Math., 29 (1980), pp. 293–309.

[Sh 80] Y. SHILOACH, *A polynomial solution to the undirected two path problem*, J. Assoc. Comput. Mach., 27 (1980), pp. 445–456.

[TV 85] R. E. TARJAN AND U. VISHKIN, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput., 14 (1985), pp. 862–874.

[Va 89] V. V. VAZIRANI, NC *Algorithms for computing the number of perfect matchings in $K_{3,3}$-free graphs and related problems*, Inform. Comput., 80 (1989), pp. 152–164.

[Wa 68] M. WATKINS, *On the existence of certain disjoint arcs in graphs*, Duke Math. J., 35 (1968), pp. 321–346.

[Wh 33] H. WHITNEY, *2-isomorphic graphs*, Amer. J. Math., 55 (1933), pp. 245–254.

[Wo 90] G. WOEGINGER, *A simple solution to the two paths problem in planar graphs*, Inform. Process. Lett., 36 (1990), pp. 191–192.

SIAM J. COMPUT.
Vol. 21, No. 3, pp. 507–520, June 1992

© 1992 Society for Industrial and Applied Mathematics
006

ALGORITHMS FOR SPLICING SYSTEMS*

R. W. GATTERDAM†

Abstract. In [*Bull. Math. Biol.*, 49 (1987), pp. 737–759] T. Head proposed a mathematical abstraction of the effect of restriction enzyme digestion and subsequent religation in the recombination of DNA molecules. In particular, he was concerned with the structure of the language of all DNA which could be produced by a *splicing system* consisting of a finite set of initial DNA and two finite sets of enzymes. He was able to show that, under appropriate conditions, the language produced is strictly locally testable and hence recursively decidable.

In this paper, an algorithmic reduction on the set of patterns is given and additional conditions introduced. The main result is that for a particular class of splicing systems there is a universal algorithm which produces a finite automaton that accepts the splicing language. The relation between this class and that studied by Head is examined and the questions of membership, finiteness, and equivalence of languages are shown to be recursive. Algorithms are given which are implementable and may be of practical interest.

Key words. DNA, finite automata, splicing system, strictly locally testable

AMS(MOS) subject classifications. 68Q20, 68Q45, 92A90

1. Introduction. The biological rationale and fundamental definitions for splicing systems are given in some detail by Head in [5]. For the sake of completeness we give a brief account here. See also Watson, Tooze, and Kurtz [9] for a general reference on DNA.

A deoxyribonucleic acid molecule (DNA) will be viewed as a string on an alphabet of four symbols, each symbol representing a deoxyribonucleotide pair. In the biological notation the pairs are A/T, C/G, G/C, and T/A. A DNA string is thought of as being oriented and the reversal of such a string is a different string. Only linear DNA, not circular, will be considered. To illustrate the biology, consider a string

$$\cdots\mathtt{CTAGAATTCGTA}\cdots$$
$$\cdots\mathtt{GATCTTAAGCAT}\cdots.$$

The action of an enzyme is to "recognize" a certain pattern, for example

$$\mathtt{GAATTC}$$
$$\mathtt{CTTAAG}$$

in the above, and to cut the string into two fragments

$$\cdots\mathtt{CTAG} \qquad\qquad \mathtt{AATTCGTA}\cdots$$
$$\cdots\mathtt{GATCTTAA} \qquad\qquad \mathtt{GCAT}\cdots.$$

Note that the G/C on the left of the pattern and C/G on the right are not involved in the cut. They are, however, required (by this particular enzyme) for the cut to take place. The staggered ends of the DNA reconnect to any matching ends so as to produce a complete DNA molecule. So for example if

$$\mathtt{AATTACT}\cdots$$
$$\mathtt{TGA}\cdots$$

*Received by the editors May 25, 1987; accepted for publication (in revised form) June 4, 1991.
†Department of Mathematical Sciences, University of Alaska, Fairbanks, Alaska 99775-1110.

had been cut from some other DNA molecule (by a different enzyme) then a new DNA string

```
···CTAGAATTACT···
···GATCTTAATGA···
```

can be formed by the recombination (called religation). To abstract this process each pair is viewed as a single symbol in an alphabet A. An enzyme is represented by a triple (a, x, b) for a,x,b in A^*. The cut and recombination takes place at x. In the above example, $a =$G/C, $x =$A/T A/T T/A T/A, and $b =$C/G. Thus if (c, x, d) represents another enzyme and $uaxbv$ and $wcxdz$ in A^* represent DNA molecules, the molecules represented by $uaxdz$ and $wcxbv$ are said to have been formed by splicing at x. We need not consider the symbol pairing aspect explicitly since we will not be concerned with DNA fragments which have not been recombined.

The formal definition of a splicing system which abstracts the concepts described above as well as other fundamental definitions will be given in §2. In §3 a reduction process on sets of enzymes is introduced and its basic properties, including associated algorithms, demonstrated. The construction of the associated automata and fundamental algorithms for splicing systems are presented in §4. In §5 the relation of this work to that of Head is explored. Finally, in §6 the results are summarized, some open questions are posed, and some applications are discussed.

2. Fundamental definitions. In this section we give the formal definition of a splicing system and also of certain terms which will be used to describe properties of such systems. Before doing so we must introduce one other biological complexity. In the example of §1, the DNA was cut

```
···G          AATTC···
···CTTAA          G···
```

but (by a different enzyme) it could be cut

```
···GAATT          C···
···C          TTAAG···.
```

Because of the orientation of the molecules, cuts of the former type cannot recombine with cuts of the latter type. Consequently two sets of patterns are introduced, called left- and right-handed.

DEFINITION 2.1. A *splicing system* $S = (A, I, B, C)$ consists of a finite alphabet A, a finite set I of *initial strings* in A^*, and finite sets B and C of triples (a, x, b) with a, x, and b in A^*. Each such triple in B or C is called a *pattern*. For each pattern (a, x, b) the string axb (often as a substring) is called a *site* and the string x is called a *crossing*. Patterns in B are called *left patterns* and patterns in C are called *right patterns*. For $uaxbv$ and $wcxdz$ in A^* with (a, x, b) and (c, x, d) patterns of the same hand, $uaxdz$ and $wcxbv$ are constructed by *splicing* at the crossing x. The *language generated* by S, denoted $L(S)$, is the minimal subset of A^* which contains I and is closed under the operation of splicing. A language L is a *splicing language* if $L = L(S)$ for some splicing system S.

Example 2.1. Let $A = \{a, b, c, d, p, q, u, v, w, x\}$, $I = \{uaxbvaxbw, pcxdq\}$, $B = \{(a, x, b), (c, x, d)\}$, and C be empty. The pattern (a, x, b) together with the initial string

$uaxbvaxbw$ can be used to create strings of the form $uax(bvax)^+bw$. Applying the pattern (c, x, d) and the initial string $pcxdq$, we see that the language generated is $(uax + pcx)(bvax)^*(dq + bw)$.

Notation. For u,v,w in A^*:

1. Λ denotes the empty string.
2. $u \prec w$ denotes u a substring of w, $u \neq w$.
3. $u \preceq w$ denotes u a substring of w, possibly $u = w$.
4. $u \dashv w$ denotes u a terminal substring (suffix) of w, i.e., $w = vu$, possibly $u = w$.
5. $u \vdash w$ denotes u an initial substring (prefix) of w, i.e., $w = uv$, possibly $u = w$.

DEFINITION 2.2. A splicing system $S = (A, I, B, C)$ is *crossing disjoint* if there do not exist patterns (a, x, b) in B and (c, x, d) in C with the same crossing x. In the case $x = \Lambda$ we assume all patterns (a, Λ, b) are in B.

Convention. All splicing systems are assumed crossing disjoint. In the following X always denotes $B \cup C$ for a crossing disjoint splicing system and we simply write $S = (A, I, X)$.

In the full generality of splicing systems there can be a great deal of interaction between various splicing occurrences. For example, it is possible for a crossing of a site y to appear in a splicing product $uaxdz$ formed from $uaxbv$ and $wcxdz$ as in Definition 2.1 which did not appear in either $uaxbv$ or $wcxdz$. Similarly sites appearing in $uaxbv$ or $wcxdz$ can be destroyed in $uaxdz$. For most of this work we limit this aspect of the complexity with the following conditions.

DEFINITION 2.3. X is *permanent* if for each pair of strings $uaxbv$, $wcxdz$ in A^* with (a, x, b) in X, (c, x, d) in X : If y is a subsegment of uax (respectively, xdz) that is a crossing of a site in $uaxbv$ (respectively, $wcxdz$) then the **same** subsegment y of $uaxdz$ is a crossing of a site in $uaxdz$.

Example 2.2. Let $A = \{a, b, c, d, u, v, w, x, y\}$, $I = \{udbcabcv, abcvwabcd, vabxabcy\}$, and patterns $B = \{(\Lambda, ab, \Lambda), (\Lambda, bc, \Lambda), (\Lambda, d, \Lambda)\}$ and C be empty. In this example there is no context (called *null context*) so $X = B \cup C = B$ is necessarily permanent. In §4 the language for this example will be computed.

DEFINITION 2.4. X is *full context* if when (a, x, b) and (c, x, d) are in X then (a, x, d) is in X.

Note that since X is $B \cup C$ for a crossing disjoint splicing system, the patterns of the crossings x of Definitions 2.3 and 2.4 are all of the same hand. Also observe that Example 2.1 is not full context but Example 2.2 (being null context) is full context.

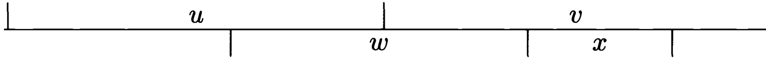
Example 2.3. Let $A = \{a, b, c, d, x, y\}$, $B = \{(a, xy, b), (c, xy, d), (ax, y, d), (c, x, yb)\}$, and C be empty. Then $X = B \cup C = B$ is not permanent since splicing $axyb$ with $cxyd$ at xy yields $axyd$ in which the substring xy is no longer a crossing. Note that this occurs because X is not full context.

Example 2.4. Patterns can be added to those of Example 2.3 so as to make X full context. Let $B = \{(a, xy, b), (c, xy, d), (a, xy, d), (c, xy, b), (ax, y, d), (c, x, yb)\}$. Note that X is not permanent since splicing $axyd$ with $cxyb$ at xy yields $axyb$ with y a site in $axyd$ but not in $axyb$.

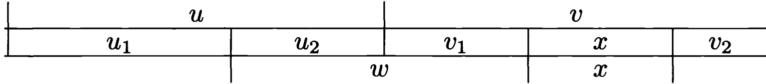
Since the sets B and C are finite, it is certainly algorithmic to determine if S is crossing disjoint. Moreover it is algorithmic to determine if X is permanent since the u,v,w,z of Definition 2.3 need be no longer than the maximum length of eyf for (e, y, f) in X. Thus a finite search can be used to determine if X is permanent. A more efficient algorithm is given in §3.

The fundamental results of §§3, 4, and 5 depend on analysis of substrings in DNA strings. The location of the substrings will be as important as their spelling and so we

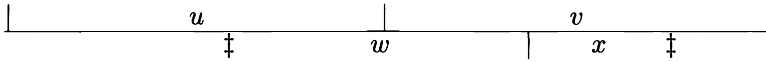
adopt the following notation. The diagram



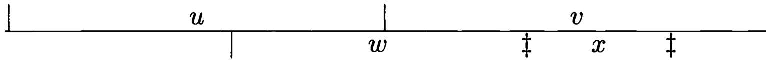
is used to indicate that $wx \prec uv$ with $x \prec v$ and that there exist $u_1 \neq \Lambda$, $u_2 \neq \Lambda$, $v_1 \neq \Lambda$, $v_2 \neq \Lambda$, such that $u = u_1u_2$, $v = v_1xv_2$, $u_2v_1 = w$, and $uv = u_1u_2v_1xv_2 = u_1wxv_2$. That is,



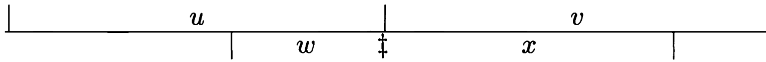
is implied. The point is that the u_1, u_2, v_1, v_2 will play no role except to assure strict string containment. Similarly,



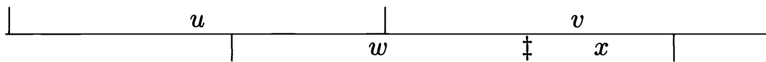
has the same meaning except that u_1 and/or v_2 may be Λ. Also



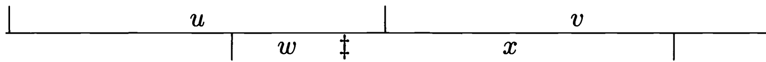
means $u_1 \neq \Lambda$, but v_1 and/or v_2 may be Λ. Finally,



means that the left end of x (right end of w) may be anywhere within u or v. That is either



or



is possible.

3. Reduction. In the following $S = (A, I, B, C)$ is a crossing disjoint splicing system and $X = B \cup C$. An algorithmic reduction of X will be demonstrated, the purpose of which is to remove certain superfluous patterns without changing the language $L(S)$. The intuition is that for X permanent there can be only limited interaction between splicings with various patterns. Moreover only trivial interaction is possible and that can be removed by the reduction process.

DEFINITION 3.1. *Reduction* is the process of forming a subset X_0 of X by removing patterns from X according to the rules:

1. If (a, x, b) and (p, x, q) are in X such that $p \dashv a$, $q \vdash b$, and $p \neq a$ or $q \neq b$, then remove (a, x, b) from X.

2. If x and y are crossings with $x \prec y$, say, $y = y_1 x y_2$, and for each (r, y, s) in X there is (p, x, q) in X with $p \dashv r y_1$ and $q \vdash y_2 s$ (i.e., each pattern with crossing y contains a pattern with crossing x inside y) then remove all patterns with crossing y from X.

DEFINITION 3.2. A set of patterns is *reduced* if no patterns of X are removed by reduction (i.e., $X_0 = X$).

The pattern sets of the examples in §2 are all reduced. Note in particular Example 2.3.

PROPOSITION 3.1. *The set X_0 is independent of the order of the removals according to rules* 1 *and* 2 *of Definition* 3.1.

Proof. Removals of type 1 cannot affect removals of type 2 since (p, x, q) is removed only if (a, x, b) is in X and $axb \prec pxq$. Thus if (p, x, q) could be used in a type 2 removal to verify that a crossing y can be removed, the site axb works as well. In the same way, if a crossing x can be used in a type 2 removal to remove a crossing y and a crossing z used to remove x, then z can be used to remove y. □

PROPOSITION 3.2. *The reduction process is algorithmic.*

Proof. It is necessary to check finitely many substrings of finitely many strings against a finite set of strings. □

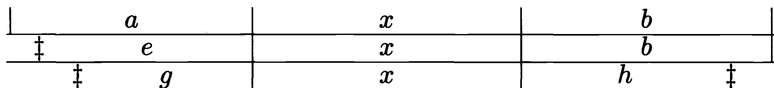
For X_0 obtained from X by reduction, $X_0 = B_0 \cup C_0$, where B_0 and C_0 are, respectively, the left and right patterns in X_0 (i.e., $B_0 = X_0 \cap B$ and $C_0 = X_0 \cap C$). Moreover (A, I, B_0, C_0) is a crossing disjoint splicing system. Reduction does not affect the language generated by the splicing system as formally stated below.

THEOREM 3.1. $L(A, I, X) = L(A, I, X_0)$.

Proof. If (a, x, b) and (p, x, q) are as in rule 1 of Definition 3.1, then any word in $L(A, I, X)$ constructed by a splice using axb could be constructed by a splice using pxq and so is in $L(A, I, X)$. Similarly, if (r, y, s) and (p, x, q) are as in rule 2 then any word constructed by a splice using rys could be constructed by a splice using pxq. □

PROPOSITION 3.3. *If X is permanent and reduced then X is full context.*

Proof. Suppose (a, x, b) and (c, x, d) are in X. Splicing axb with cxd yields axd. Since X is permanent, x is the crossing of a site in axd, say, (e, x, f) in X with $e \dashv a$, $f \vdash d$. Crossing exf with axb yields exb. Then in exb, x is the crossing of a site (g, x, h) in X with $g \dashv e \dashv a$, $h \vdash b$ as shown:



Thus if $e \neq a$, X can be reduced by removing (a, x, b), i.e., by a type 1 reduction. Since X is reduced, $a = e$. By a similar argument, $f = d$ so (a, x, d) is in X. □

PROPOSITION 3.4. *If X is permanent then any splice is reversible.*

Proof. If uxv is spliced with wxz at x to produce uxz and wxv, the x remains the crossing of a site in uxz and wxv, so uxv and wxz can be recovered by splicing uxz and wxv at x. □

The significance of permanent and reduced is seen in the next theorem.

THEOREM 3.2. *If X is permanent and reduced then splicing can neither create nor destroy crossings. That is, if $upxqv$ and $waxbz$ are such that (p, x, q) and (a, x, b) are in X, then y is the crossing of a site in $upxbz$ or $waxqv$ if and only if y is the crossing of a site in $upxqv$ or $waxbz$ at the same location.*

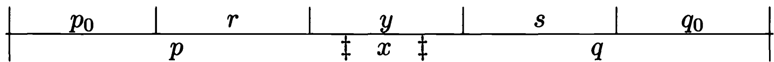
Proof. With the notation as above, we first show that y a crossing of a site in $upxqv$ implies y a crossing of a site in $upxbz$ or $waxqv$. It suffices to consider the case $x \prec y$, since if $y \prec upx$ or $y \prec xqv$, then y is a crossing of a site in $upxbz$ or $waxqv$, since X is permanent. Thus let (r, y, s) be in X with $rys \preceq upxqv$ and $x \prec y$. We proceed by cases to show $x \prec y$ $(x \neq y)$ is a contradiction to X being permanent and reduced.

1. $r \prec p$.

1.1. There exist $p_0 \neq \Lambda$ and $s_0 \neq \Lambda$ such that $p_0rys = pxqs_0$,

Splicing p_0rys with rys at y yields rys with x the crossing of a site in p_0rys, hence in rys since X is permanent. Therefore (p_1, x, q_1) is in X with $p_1 \dashv p$. Since X is reduced, $q \prec q_1$ as otherwise $p_1xq_1 \preceq pxq$, and X can be reduced by removing (p, x, q). Then splicing p_1xq_1 with pxq at x yields $p_1xq \prec pxq$, with x the crossing of a site $p_2xq_2 \preceq p_1xq \preceq pxq$, contrary to X being reduced. Therefore case 1.1 cannot occur.

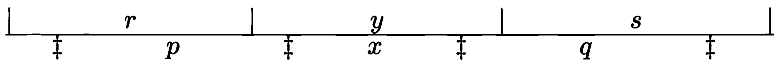
1.2. There exists $p_0 \neq \Lambda$ and $q_0 \neq \Lambda$ such that $p_0rysq_0 = pxq$,



Splicing p_0rysq_0 with rys at y yields $rysq_0$ and splicing again with rys at y yields rys. Since x was the crossing of a site in p_0rysq_0 it must be a crossing of a site in rys because X is permanent. Thus $p_1xy_1 \preceq rys \prec pxq$ with (p_1, x, y_1) in X contrary to X being reduced. Therefore case 1.2 and hence case 1 cannot occur.

2. $s \prec q$. By an argument similar to that for case 1, case 2 cannot occur.

3. $pxq \preceq rys$ (this includes the possibility $pxq \preceq y$),



We show that every site (m, y, n) in X contains a site with crossing the indicated $x \prec y$. Splice myn and rys at y to obtain ryn and mys. In each, x remains the crossing of a site. Splice ryn with mys at y to obtain myn. Again x must remain the crossing of a site in myn. Since the above applies to every such (m, y, n), X can be reduced by a type 2 reduction applied to y. Since X is reduced case 3 cannot occur.

Since none of $r \prec p$, $s \prec q$, or $pxq \preceq rys$ can occur, no crossing y with $x \prec y$ can be destroyed. Thus crossings of sites cannot be destroyed. To see that crossings of sites cannot be created, we observe that if X is permanent any splice is reversible by Proposition 3.4. Thus if $upxqv$ and $wrxsz$ are spliced at x to produce $upxsz$ and $wrxqv$, x remains the crossing of a site in each so they can be spliced at x to recover $upxqv$ and $wrxsz$. Therefore if the original splice created a site, the resplice would destroy it, which is not possible by the preceding argument. □

By definition, if X is not permanent then crossings of sites can be destroyed, so for X reduced, permanent is equivalent to crossings of sites not being destroyed or created by splicing.

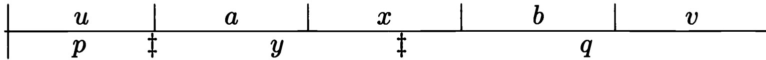
The remainder of this section is devoted to the development of an algorithm to determine if a crossing disjoint, reduced splicing system is permanent. The existence of such an algorithm for an arbitrary splicing system is apparent as observed in §2 but after

reduction, a simplified algorithm is possible. The details of this development are not required in subsequent sections and the reader may skip to §4.

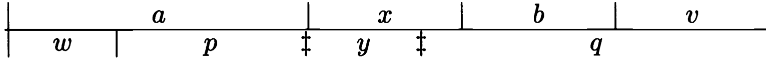
PROPOSITION 3.5. *Assume X is reduced and full context (as is the case if X is reduced and permanent). Then for (a, x, b) in X, there does not exist $(a'a, x, d)$ in X for $a' \neq \Lambda$. Similarly, there does not exist (c, x, bb') in X for $b' \neq \Lambda$.*

Proof. If $(a'a, x, d)$ is in X, then since X is full context, (a, x, d) is in X with $axd \prec a'axd$, so $a' = \Lambda$ or X is not reduced. The second statement is proved in a similar way. □

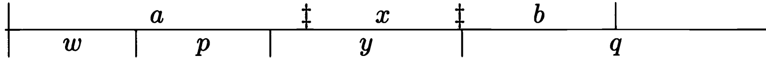
PROPOSITION 3.6. *Assume X is reduced and permanent with (a, x, b) and (p, y, q) in X. If either $uaxbv = pyq$ with $y \prec uax$ or $axbv = wpyq$ with $y \prec ax$ for some u, w then $v = \Lambda$. That is, if*



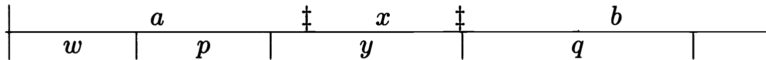
or



then $v = \Lambda$. Similarly, if

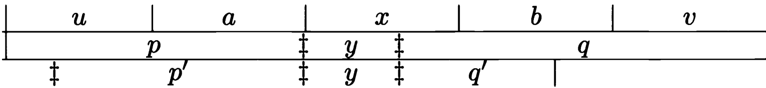


or



then $w = \Lambda$.

Proof. Splicing $uaxbv$ (or $axbv$) with axb at x yields $uaxb$ (or axb). Since X is permanent, there exists (p', y, q') in X so that



If $v \neq \Lambda$, then $q' \preceq xb$ with $b \neq q$, so $q' \prec q$ with $q' \neq q$, contrary to Proposition 3.5. The second statement is proved in a similar way. □

THEOREM 3.3. *Let X be a reduced set of patterns. The following algorithmic tests are to be performed in sequence:*

1. Determine if X is full context.
2. Determine if there do not exist (a, x, b) and (c, y, d) in X with $cyd \preceq uaxbv$ and either

    2a. $cyd \dashv uaxbv$, $cy \preceq uax$ and $bv \dashv d$ with $v \neq \Lambda$ or

    2b. $cyd \vdash uaxbv$, $yd \preceq xbv$ and $ua \vdash c$ with $u \neq \Lambda$.

3. Determine if there do not exist (a, x, b) and (c, y, d) in X with $cyd \preceq uaxbv$ and either

3a. $d = d_0 d_1$, $cyd_0 \dashv uax$, $d_1 w = b$ and there exists (a, x, b') in X such that for every writing $b' = d'w'$, $(c, y, d_0 d')$ does not belong to X (note v plays no role) or

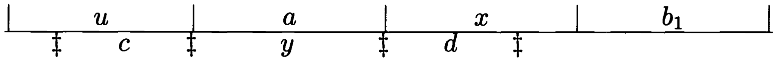
3b. $c = c_1 c_0$, $c_0 yd \vdash xbv$, $wc_1 = a$ and there exists (a', x, b) in X such that for every writing $a' = w'c'$, $(c'c_0, y, d)$ does not belong to X (note u plays no role).

If X has passed the above sequence of tests, then X is permanent.

Proof. First, X must be full context by Proposition 3.4. Assume X is full context. In order for X to be permanent, we must have that for any splice using patterns (a_1, x, b_1) and (a_2, x, b_2) in X for which (c, y, d) is in X and $cyd \preceq ua_1xb_1v$ with $cy \preceq ua_1x$, there exists (c', y, d') in X such that $c'yd' \preceq ua_1xb_2$ (i.e., y remains the crossing of a site after ua_1xb_1v and a_2xb_2 have been spliced at x to produce ua_1xb_2). Similarly, if $cyd \preceq ua_1xb_1v$ with $yd \preceq xb_1v$ then there must exist (c', y, d') in X such that $c'yd' \preceq a_2xb_1v$. In all cases the y must be the same subsegment. If the above conditions hold for every choice (a_1, x, b_1), (a_2, x, b_2), and (c, y, d) in X then X is permanent.

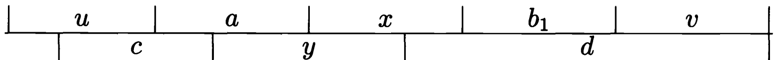
We next investigate the situation (a_1, x, b_1), (a_2, x, b_2), and (c, y, d) in X with $cyd \preceq ua_1xb_1v$ and $cy \preceq ua_1x$. Since X is full context we may take $a_1 = a_2 = a$. There are three cases to consider:

1. $cyd \preceq uax$, $yd \not\preceq x$,



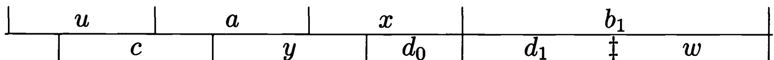
In this case $cyd \preceq uaxb_2$. That is, this case cannot result in a situation so that X is not permanent.

2. $cyd \dashv uaxb_1v$, $b_1v \dashv d$ for $v \neq \Lambda$,



By Proposition 3.6, this case implies X is not permanent.

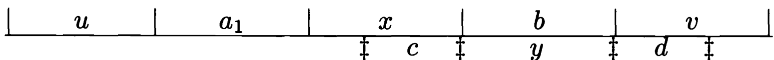
3. $d = d_0d_1$ so that $cyd_0 \dashv uax$ and $d_1w = b_1$,



Then for every (a, x, b_i) in X, splicing $uaxb_1$ with axb_i for $i \geq 2$ we see that for X to be permanent it is necessary that there exists (c, y, d_0d_i) in X so that $d_i \vdash b_i$. Note that (c_i, y, d_0d_i) in X is sufficient but since X is full context we may take each c_i to be c.

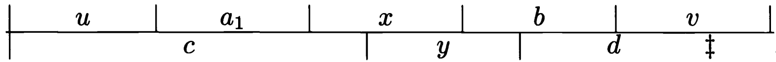
Similarly, if (a_1, x, b), (a_2, x, b), and (c, y, d) are in X with $cyd \preceq ua_1xbv$ and $yd \preceq xbv$ then the cases to be considered are:

4. $cyd \preceq xbv$, $cy \not\preceq x$,



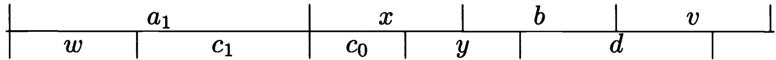
This situation cannot result in X being not permanent.

5. $cyd \vdash ua_1xbv$, $ua_1 \vdash c$ for $u \neq \Lambda$,



This situation implies X not permanent by Proposition 3.6.

6. $c = c_1c_0$ so that $c_0yd \vdash xbv$ and $wc_1 = a_1$,



For X to be permanent in this situation it is necessary that for every (a_i, x, b_i) in X there exists (c_ic_0, y, d) in X so that $c_i \dashv a_i$. □

4. Associated automaton. Let $S = (A, I, X)$ be a crossing disjoint, reduced, permanent splicing system. We construct a nondeterministic finite automaton from S called the *associated automaton* and denoted $\Psi(S)$ as follows:

1. Let $I = \{I_1, \cdots, I_m\}$ and let $len(i) = len(I_i)$ denote the length (number of symbols) of I_i. Then construct states $\langle i, j\rangle$ for $i = 1, \cdots, m$ and $j = 0, \cdots, len(I_i)$. Each $\langle i, 0\rangle$ is an initial state and each $\langle i, len(i)\rangle$ is a terminal state. State transitions are $\langle i, j-1\rangle$ to $\langle i, j\rangle$ on symbol a in A if a is the jth symbol in I_i.

2. If x is the crossing of patterns (u, x, v) in X and uxv is a site in I_i with the final character of x the jth character of I_i, then label $\langle i, j\rangle$ as Px. Associate all states labeled Px (i.e., collapse all states labeled Px into a single state). Note that the association of states depends only on x and not the particular pattern (u, x, v). The state Px is called the *crossing state* for x. We denote by $L(\Psi(S))$ the language accepted by $\Psi(S)$.

The definition of $\Psi(S)$ is a uniform algorithm for its construction from S. The intuition of the construction of $\Psi(S)$ is that each initial string is viewed as a linear automaton from initial to acceptance state and the sites are then "glued together" to form the final automaton. An automaton produced by an application of the definition is of course nondeterministic in general but by the usual technique (e.g., in [7]), a deterministic automaton accepting the same language can be constructed. The main result is contained in the following theorem.

THEOREM 4.1. *Let $S = (A, I, X)$ be a crossing disjoint, reduced, permanent splicing system and $\Psi(S)$ be its associated automaton. Then $L(S) = L(\Psi(S))$.*

Proof. Rather than deal with a word of $L(\Psi(S))$ directly, it will be convenient to consider an *acceptance path* in $\Psi(S)$. By this we will mean a sequence of states and transitions in $\Psi(S)$ beginning at an initial state and ending at a terminal state. An acceptance path will be described by a sequence of words $w_1, \cdots, w_{m+1}$ in A^* and crossing states $Px_1, \cdots, Px_m$ such that:

1. $w_1 \vdash I_{i(1)}$ and there is a sequence of states starting at $\langle i(1), 0\rangle$ and ending at Px_1 with transitions corresponding to the symbols of w_1.

2. For $1 < j < m+1$, $w_j \preceq I_{i(j)}$ and there is a sequence of states starting at Px_{j-1} and ending at Px_j with transitions corresponding to the symbols of w_j.

3. $w_{m+1} \dashv I_{i(m+1)}$ and there is a sequence of states starting at Px_m and ending at $\langle i(m+1), len(i(m+1))\rangle$ with transitions corresponding to the symbols of w_{m+1}.

4. w_j and w_{j+1} are not contiguous substrings of the same $I_{i(j)}$. Note however that w_j and w_{j+1} may be in the same $I_{i(j)} = I_{i(j+1)}$. Intuitively this condition guarantees that a splicing takes place at Px_j.

A word w in A^* is in $L(\Psi(S))$ if and only if $w = w_1 \cdots w_{m+1}$ for $w_1, \cdots, w_{m+1}$ and corresponding Px_j an acceptance path.

LEMMA 4.1.1. *If $w_1, \cdots, w_{m+1}$ for $w_j \prec I_{i(j)}$ and $Px_1, \cdots, Px_m$ is an acceptance path, then $w = w_1 \cdots w_{m+1}$ can be constructed by crossing $I_{i(1)}$ with $I_{i(2)}$ at the site corresponding to Px_1, crossing the result with $I_{i(3)}$ at the site corresponding to Px_2, etc., and crossing the result with $I_{i(m+1)}$ at the site corresponding to Px_m.*

Proof. The proof is by induction on the length of the acceptance path, m. For $m = 0$, $w = w_1 = I_{i(1)}$. Assume the statement is true for acceptance paths of length $m - 1$, and consider an acceptance path $w_1, \cdots, w_{m+1}$ for $w_j \prec I_{i(j)}$ and crossing states $Px_1, \cdots, Px_m$. Then $w_1, \cdots, w_{m-1}, w_m u$ for $w_m \prec I_{i(m)} = vw_m u$ and $Px_1, \cdots, Px_{m-1}$ is an acceptance path of length $m - 1$, and $w' = w_1 \cdots w_m u$ can be constructed by $m - 1$ crossings as in the statement of the lemma. In particular, w' is in $L(S)$. Since S is permanent and reduced, crossings of sites cannot be destroyed by Theorem 3.2, so x_m is the crossing of a site in w' at the indicated Px_m. Therefore w' can be crossed with $I_{i(m+1)}$ at that crossing to produce $w = w_1 \cdots w_{m+1}$. □

By definition w is in $L(\Psi(S))$ if and only if there is an acceptance path for w in which case w is in $L(S)$ by Lemma 4.1.1. Thus $L(\Psi(S))$ is contained in $L(S)$.

LEMMA 4.1.2. *If $w_1, \cdots, w_{m+1}$ for $w_j \prec I_{i(j)}$ and $Px_1, \cdots, Px_m$ is an acceptance path, and $w = w_1 \cdots w_{m+1}$ is such that $rxs \preceq w$ for (r, x, s) in X, and if the final symbol of x is the kth symbol of w, then the $k + 1$st state of the acceptance path is Px.*

Proof. The proof is by induction on the acceptance path length m. If $m = 0$ then $w = w_1 = I_{i(1)}$, so $rxs \prec I_{i(1)}$ implies the conclusion by the construction of $\Psi(S)$. As in the proof of Lemma 4.1.1, $w_1, \cdots, w_{m-1}, w_m u$ for $w_m \prec I_{i(m)} = vw_m u$ with $Px_1, \cdots, Px_{m-1}$ is an acceptance path of length m for $w' = w_1 \cdots w_m u$. By the preceding, w' is in $L(S)$ and w is formed by crossing w' with $I_{i(m+1)}$ at x_m. Since by Theorem 3.2 the operation of splicing cannot create a crossing of a site, x must have been the crossing of a site in either w' or in $I_{i(m+1)}$. In either case, by the inductive hypothesis or definition of $\Psi(S)$, the acceptance path for w' or $I_{i(m+1)}$ contains Px as indicated and consequently so does the given acceptance path for w. □

Proof of Theorem 4.1. Let w', w'' be in $L(\Psi(S))$ with x the crossing of a site in both. Let w be formed by splicing w' and w'' at x. By Lemma 4.1.2 any acceptance path for w' contains Px and so does any acceptance path for w''. Therefore these paths can be merged at Px to produce an acceptance path for w. Thus w is in $L(\Psi(S))$ and $L(\Psi(S))$ is closed under the operation of splicing. We conclude $L(S)$ is contained in $L(\Psi(S))$ and $L(\Psi(S)) = L(S)$. □

Example 4.1. It is an easy matter to apply the technique of Theorem 4.1 to Example 2.2. The initial strings *udbcabcv*, *abcvwabcd*, and *vabxabcy* are each viewed as a linear sequence of states with transitions on the next letter of the string. That is, state $\langle 1, 0\rangle$ is an initial state with transition on u to state $\langle 1, 1\rangle$ with transition d to state $\langle 1, 2\rangle$ etc. and finally on transition v to state $\langle 1, 8\rangle$, which is a terminal state. Similarly, for the other two initial strings from state $\langle 2, 0\rangle$ to $\langle 2, 9\rangle$ and $\langle 3, 0\rangle$ to $\langle 3, 8\rangle$. Next associate (glue together) states $\langle 1, 2\rangle$ and $\langle 2, 9\rangle$ according to crossing *d*, states $\langle 1, 4\rangle$, $\langle 1, 7\rangle$, $\langle 2, 3\rangle$, $\langle 2, 8\rangle$, and $\langle 3, 7\rangle$ according to crossing *bc*, and states $\langle 1, 6\rangle$, $\langle 2, 2\rangle$, $\langle 2, 7\rangle$, $\langle 3, 3\rangle$, and $\langle 3, 6\rangle$ according to crossing *ab*. After reduction of the resulting automaton, the language is seen to be $ud + ((v + \Lambda)ab(xab)^*c + udbc)((\Lambda + vw)ab(xab)^*c + dbc)^*(y + v + d)$.

COROLLARY 4.1.1. *For $S = (A, I, X)$ a crossing disjoint, reduced, permanent splicing system, any word w in $L(S)$ can be formed by a sequence of splicing operations from "left to right," each splice being with a member of I.*

Proof. Since w is in $L(S)$ it is in $L(\Psi(S))$ by Theorem 4.1, so the statement of the

corollary is a rephrasing of Lemma 4.1.1. □

COROLLARY 4.1.2. *For $S = (A, I, X)$ a crossing disjoint, reduced, permanent splicing system it is algorithmic to determine if $L(S)$ is a finite language.*

Proof. By definition it is algorithmic to construct $\Psi(S)$. It is algorithmic to determine if the language accepted by a finite automaton is finite, so the required algorithm is to construct $\Psi(S)$ and determine if $L(\Psi(S))$ is finite. □

COROLLARY 4.1.3. *For $S = (A, I, X)$ and $S' = (A, I', X')$ crossing disjoint, reduced, permanent splicing systems it is algorithmic to determine if $L(S) = L(S')$.*

Proof. Construct $\Psi(S)$ and $\Psi(S')$ and by a standard algorithm determine if $L(\Psi(S))$ and $L(\Psi(S'))$ are equal. (That is, construct a finite automaton to accept the symmetric difference of $L(\Psi(S))$ and $L(\Psi(S'))$ and determine if the language it accepts is empty.) □

5. Persistence. In [5] Head proposed the concept of a *persistent* splicing system and showed that such a system is strictly locally testable using the results of Schutzenberger [8] and DeLuca and Restivo [2]. His result does not require that the splicing system be crossing disjoint. Since a strictly locally testable language is regular and, as we will see, permanent implies persistent, his result is stronger than that of §4. The proof that such a language is strictly locally testable is not constructive, however, in that it gives no indication of how to produce an algorithm to solve the membership problem or create a finite automaton to recognize the language based on (A, I, B, C). In that sense, §4 can be viewed as a realization of Head's result. We show below that, restricted to crossing disjoint, full context splicing systems, our results and those of Head apply to the same class of problems.

DEFINITION 5.1. A pair of left- and right-hand pattern sets B, C (not necessarily crossing disjoint) is *persistent* if for each pair of strings $uaxbv$, $wcxdz$ in A^* with (a, x, b) and (c, x, d) patterns of the same hand: If y is a subsegment of uax (respectively, xdz) that is a crossing of a site in $uaxbv$ (respectively, $wcxdz$) then the **same** subsegment y of $ucxdz$ contains a crossing of a site in $ucxdz$.

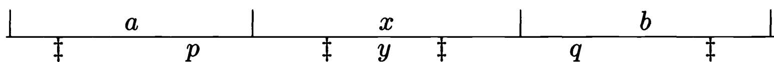
Note that if the word **contains** in the definition of *persistent* is replaced by **is** and B, C by X, then the result is Definition 2.3 of *permanent*. In particular permanent implies persistent. The converse is true for reduced, full context systems as we see below. This fact will also show that the converse to Proposition 5.1 below is false.

PROPOSITION 5.1. *If X is permanent and X_0 is formed from X by reduction, then X_0 is permanent.*

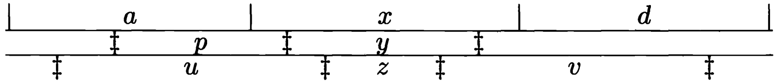
Proof. This follows immediately from Theorem 3.3 since X contains X_0. □

THEOREM 5.1. *For crossing disjoint splicing systems, a persistent, reduced, full context set of patterns is permanent.*

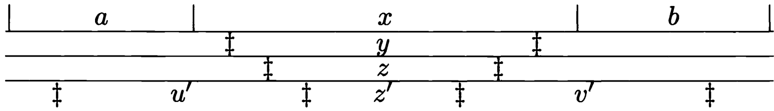
LEMMA 5.1.1. *There do not exist (a, x, b) and (p, y, q) in X such that $y \prec x$ and $pyq \preceq axb$. That is not*



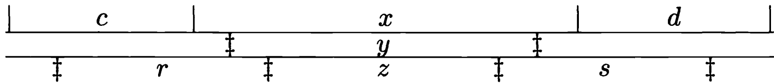
Proof. Let (a, x, b) and (p, y, q) be as in the statement and assume y minimal in that there does not exist (p', y', q') in X with $y' \prec y$ and $p'yq' \preceq axb$. We show (c, x, d) in X implies there exists (r, y, s) in X such that $y \prec x$ and $rys \prec cxd$. This is contrary to X reduced since all patterns with crossing x can then be removed from X by a type 2 reduction. Let (c, x, d) be in X and splice axb with cxd at x to produce axd. Then since X persistent, there exists (u, z, v) in X such that



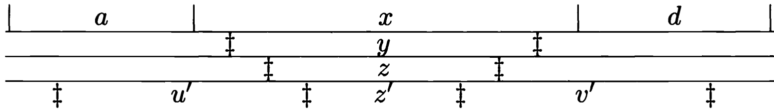
If $z \prec y$, $z \neq y$, then splicing axd with axb at x to produce axb, there exists (u', z', v') in X so that



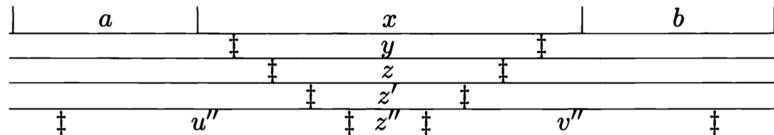
But this is contrary to the minimality of y. Therefore $z = y$. Splicing axd with cxd at x we obtain cxd and since X is persistent there exists (r, z, s) in X so that



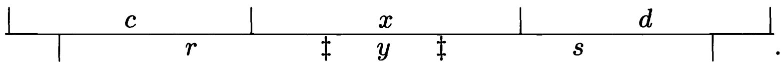
If $z \prec y$, $z \neq y$ then splicing with axb yields axd and there exists (u', z', v') in X so that



Finally, splicing with axb at x yields axb and there exists (u'', z'', v'') in X so that



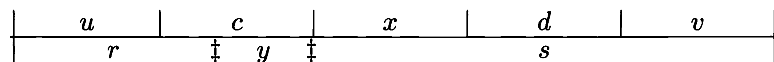
contrary to the minimality of y. Therefore $z = y$ and



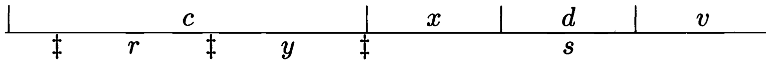
□

Proof of Theorem 5.1. We consider $ucxdv$ and $pexfq$ with (c, x, d) and (e, x, f) in X to be spliced at x. Also $rys \preceq ucxdv$, $ry \preceq ucx$, and (r, y, s) in X. We must show that y remains the crossing of a site in $ucxfq$. Without loss of generality we may take u and v minimal so one of the following pertains:

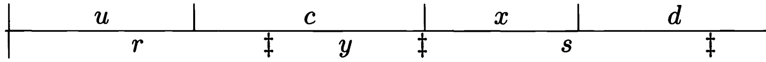
1.



2.

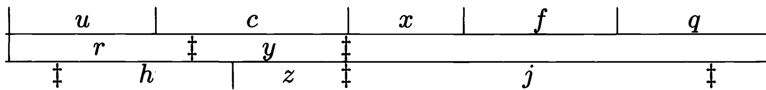


3.

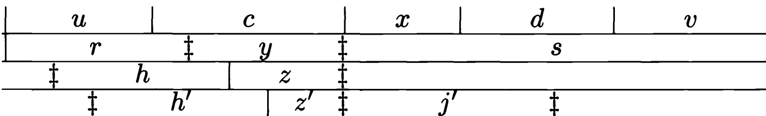


4.

We treat case 1, the others being similar. After splicing $ucxdv$ with $pexfq$ at x, we have that there exists (h, z, j) in X so that



since X is persistent. Since X is full context, (c, x, f) is in X so $ucxfq$ can be spliced with $ucxdv$ at x to produce $ucxdv$. Again since X is persistent there exists (h', z', j') in X such that



Then $h'z'j' \preceq rys$ with $z' \preceq y$ so by Lemma 5.1.1, $z' = z = y$. That is, y is the crossing of a site, (h, y, j), in $ucxfq$ as required. □

COROLLARY 5.1.1. *If X is persistent and X_0 the reduction of X is full context, then X_0 is permanent.*

Proof. Any splice that can take place by virtue of a pattern in X can take place by virtue of a pattern in X_0. Hence X persistent implies X_0 persistent, which implies X_0 permanent since X_0 is reduced. □

To see the that a nonreduced persistent system need not be permanent, it is possible to construct an example wherein a site which would be removed by a type 2 reduction does not remain a site after a splice. The interested reader may supply the details. Example 2.3 is persistent and reduced but not permanent; Example 2.4 is reduced and full context but not persistent.

We remark that crossing disjoint seems to be a necessary condition for the construction of the automaton of §4 to work, for otherwise the automaton recognizes strings formed by the illegal splicing of a left-hand pattern with a right-hand pattern. The conditions of crossing disjoint and full context (after reduction) represent a significant gap between this work and that of Head.

6. Conclusion. Given a splicing system $S = (A, I, B, C)$ the sequence of algorithms to be applied is:

1. Determine if B and C are crossing disjoint. If so $X = A \cup B$ and proceed to 2.
2. Reduce X to X_0 according to the algorithm of Definition 3.1.
3. Determine if X_0 is permanent using the algorithm given by Theorem 3.3. If so proceed to 4.
4. Construct $\Psi(S)$ according to the definition.
5. From $\Psi(S)$ answer the membership or finiteness question or, from two such applications of 1 through 4, the equivalence question.

Computer programs to implement the algorithms above have been written.

In view of Head's result, the obvious question is what can be said if S is not persistent? Culik and Harju [1] have shown that the language is still regular. In contrast, if the number of copies of a DNA type is limited, the resulting language may be recursively undecidable [3]. Reference [4] contains additional examples (including one which is not strictly locally testable) and a converse of the regularity of splicing systems valid up to homomorphism. Finally, a more extensive model of the DNA process, consistent with the methods of this paper, is proposed by Head in [6].

REFERENCES

[1] K. Culik II and T. Harju, *The regularity of splicing systems and* DNA, 16th International Colloquium on Automata Languages and Programming, Lecture Notes in Computer Science, 372, Springer-Verlag, Berlin, Heidelberg, New York, 1989, pp. 222–233.

[2] A. DeLuca and A. Restivo, *A characterization of strictly locally testable languages and its application to subsemigroups of a free semigroup*, Inform. Control, 44 (1980), pp. 300–319.

[3] K. L. Denninghoff and R. W. Gatterdam, *On the undecidability of splicing systems*, Internat. J. Comput. Math., 27 (1989), pp. 133–145.

[4] R. W. Gatterdam, *Splicing systems and regularity*, Internat. J. Comput. Math., 31 (1989), pp. 63–67.

[5] T. Head, *Formal language theory and* DNA: *An analysis of the generative capacity of specific recombinant behaviors*, Bull. Math. Biol., 49 (1987), pp. 737–759.

[6] ———, *Splicing schemes and* DNA, preprint.

[7] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison–Wesley, Reading, MA, 1979.

[8] M. P. Schutzenberger, *Sur certaines operations de fermeture dans les langages rationnels*, Sympos. Math., 15 (1975), pp. 245–253.

[9] J. D. Watson, J. Tooze, and D. T. Kurtz, *Recombinant* DNA: *A Short Course*, Freeman, New York, 1983.

SIAM J. COMPUT.
Vol. 21, No. 3, pp. 521–539, June 1992

© 1992 Society for Industrial and Applied Mathematics
007

RELATING EQUIVALENCE AND REDUCIBILITY TO SPARSE SETS*

ERIC ALLENDER†, LANE A. HEMACHANDRA‡, MITSUNORI OGIWARA§, AND OSAMU WATANABE¶

Abstract. For various polynomial-time reducibilities r, this paper asks whether being r-reducible to a sparse set is a broader notion than being r-equivalent to a sparse set. Although distinguishing equivalence and reducibility to sparse sets, for many-one or 1-truth-table reductions, would imply that $P \neq NP$, this paper shows that for k-truth-table reductions, $k \geq 2$, equivalence and reducibility to sparse sets provably differ. Though Gavaldà and Watanabe have shown that, for any polynomial-time computable unbounded function $f(\cdot)$, some sets $f(n)$-truth-table reducible to sparse sets are not even Turing equivalent to sparse sets, this paper shows that extending their result to the 2-truth-table case would provide a proof that $P \neq NP$. Additionally, this paper studies the relative power of different notions of reducibility, and proves that disjunctive and conjunctive truth-table reductions to sparse sets are surprisingly powerful, refuting a conjecture of Ko.

Key words. polynomial-time reducibilities, sparse sets, computational complexity, degrees

AMS(MOS) subject classifications. 68Q15, 03D30, 03D55

1. Introduction. Computer science is the study of information—coding information, decoding information, organizing information, and accessing information. Sets whose information content is small, intuitively the structurally simplest of sets, have played a central rôle in the development of the theory of computing. Sparse sets—sets with at most polynomially many elements of each length—are one natural notion of "sets of small information content," and, indeed, sparse sets have been essential to recent advances in computational complexity theory ([HM80], see also [Mah86], [Mah89]).

However, in complexity theory it is common to investigate notions sufficiently robust so as to be invariant under polynomial-time reductions. Thus, an even more natural notion of "small information content" is R^p_T(SPARSE), the class of sets that polynomial-time Turing reduce to sparse sets.[1] The sense in which R^p_T(SPARSE) sets are of small information content can be crisply formalized: R^p_T(SPARSE) is precisely the class—more commonly referred to as P/poly—of sets having polynomial-sized (nonuniform) circuits (Meyer, see [BH77]).

R^p_T(SPARSE) has been intensely studied, both in terms of the question "NP $\subseteq$ R^p_T(SPARSE)?" [KL80], [IM89], [CGH+89], [Kad89], and in terms of the robustness of R^p_T(SPARSE). R^p_T(SPARSE) is indeed quite robust; in addition to its characterization in terms of small circuits, R^p_T(SPARSE) is easily noted equivalent to R^p_T(TALLY), R^p_{tt}(TALLY), and R^p_{tt}(SPARSE) (see [BDG88]). Nonetheless, Book and Ko showed that there were limits to the robustness of R^p_T(SPARSE); they initiated the study of the classes of languages reducible to sparse (and tally) sets under various weak notions of polynomial-time reducibility, and proved that such classes differed both from

*Received by the editors October 22, 1990; accepted for publication June 25, 1991.

†Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903. The research of this author was supported in part by National Science Foundation grant CCR-9000045 and by International Information Science Foundation grant 90-1-3-227.

‡Department of Computer Science, University of Rochester, Rochester, New York 14627. The research of this author was supported in part by National Science Foundation grants CCR-8996198 and CCR-8957604, and by International Information Science Foundation grant 90-1-3-228.

§Department of Computer Science, University of Electro-communications, Tokyo 182, Japan. This research was done in part while this author was at the Tokyo Institute of Technology.

¶Department of Computer Science, Tokyo Institute of Technology, Tokyo 152, Japan.

[1]Though formal definitions will be given in §2, it is useful to introduce some notation here. For a given reducibility $\leq^p_r$, we define: (1) R^p_r(SPARSE) as the class of sets L such that, for some sparse set S, $L \leq^p_r S$, and (2) R^p_r(TALLY) as the class of sets L such that, for some tally set T, $L \leq^p_r T$.

R_T^p(SPARSE) and from each other ([BK88], see also the earlier related work on two subclasses of R_T^p(SPARSE): "almost polynomial time" [MP79] and the P-close sets [Yes83], [Sch86a]).

Tang and Book initiated an analogous study of the classes of languages *equivalent* to sparse (and tally) sets under various notions of polynomial-time reducibility, and proved that in many cases such classes differed from each other [TB]. Additionally, equivalence has been used by Balcázar and Book to characterize completely a natural subset of R_T^p(SPARSE), namely the sets with self-producible circuits [BB86].

The study of equivalence to sparse sets and the study of reducibility to sparse sets have each yielded a flurry of results [BK88], [TB], [CGH+89], [IM89], [Kad89], [Ko89], [AW90], [AH], [Ko]. Nonetheless, many of the most basic questions have remained unanswered and, in some cases, unasked.

In particular, the relationships between equivalence and reducibility to sparse sets have remained wholly unknown. The first results along this line are those of the present paper and the companion paper of Gavaldà and Watanabe (see Theorem C below). The present paper asks, for the case of bounded truth-table reductions, whether reducibility to sparse sets is a broader notion than equivalence to sparse sets. We provide answers to this question and indicate some areas in which further progress is unlikely until long-standing open problems in complexity theory are resolved. Among our results are the following.[2]

THEOREM A. R^p_{2-tt}(SPARSE) $\not\subseteq E^p_{f(n)-tt}$(SPARSE), *for any* $f(n) = n^{o(1)}$.

THEOREM B. *If* P = NP *then* R^p_{1-tt}(SPARSE) = E^p_{1-tt}(SPARSE). *If* P = NP *then* R^p_m(SPARSE)= E^p_m(SPARSE) $\bigcup\{\Sigma^*\}$.

Theorem A implies that reducibility and equivalence to sparse sets differ sharply for $\leq^p_{2-tt}$ reductions. In contrast, Theorem B indicates that proving an analogous result for $\leq^p_{1-tt}$ or $\leq^p_m$ reductions would involve proving P $\neq$ NP. Theorem A raises the issue of the strength of reducibility that will suffice to provide equivalence to some sparse set for sets bounded truth-table equivalent to sparse sets. That is, what is the cost paid—in terms of increase in strength of reduction—to achieve equivalence? Gavaldà and Watanabe have shown that for truth-table reductions whose number of queries is unbounded, an extremely heavy price is exacted. We show that the Gavaldà–Watanabe result cannot be extended to the 2-truth-table case without providing a proof that P $\neq$ NP.

THEOREM C [GW]. *Let* $f(n)$ *be any unbounded polynomial-time computable function. Then* $R^p_{f(n)-tt}$(SPARSE) $\not\subseteq E^p_T$(SPARSE).

THEOREM D. *Let* $\epsilon > 0$. *If* P = NP *then* R^p_{2-tt}(SPARSE) $\subseteq E^p_{n^\epsilon-tt}$(SPARSE).

Theorem D shows that Theorem A is optimal, and provides an upper bound, conditioned upon the assumption that P = NP, for the complexity of equivalence. We also provide unconditional upper bounds.

Finally, we turn to one of the key open questions about the structure of R^p_r(SPARSE) classes. Ko [Ko89, p. 65] conjectures that for each $k > 0$ it holds that R^p_{k-tt}(SPARSE) $\not\subseteq R^p_{dtt}$(SPARSE). We refute this conjecture by proving the following result.

THEOREM E. R^p_{btt}(SPARSE) $\subseteq R^p_{dtt}$(SPARSE).

This may be interpreted as saying that disjunctive truth-table reductions to sparse sets are surprisingly powerful. We prove related results showing that the power of conjunctive

[2]Notation: given a notion of reducibility $\leq^p_r$, we define: (1) E^p_r(SPARSE) as the class of sets L such that, for some sparse set S, $L \leq^p_r S$ and $S \leq^p_r L$, and (2) E^p_r(TALLY) as the class of sets L such that, for some tally set T, $L \leq^p_r T$ and $T \leq^p_r L$. An $f(n)$–tt reduction is a truth-table reduction that, on inputs of size n, generates at most $f(n)$ queries. See §2 for full definitions of these and other notions.

truth-table and many-one reductions, in the nondeterministic model of Ladner, Lynch, and Selman [LLS75], is also substantial.

THEOREM F.

1. R_m^{NP}(SPARSE)=R_{btt}^{NP}(SPARSE).
2. R_{ctt}^{NP}(SPARSE)=R_T^{NP}(SPARSE).

The paper is organized as follows. Section 2 reviews notations and definitions. Section 3 studies the relationship between reductions and equivalences for the cases of many-one and 1-truth-table reducibilities. Section 4 studies the case of k-truth-table reductions, $k \geq 2$—a case that differs sharply from those of §3. Section 5 investigates the interrelationships between reducibility classes and their seemingly restrictive (but surprisingly powerful) disjunctive and conjunctive versions. Section 6 presents open problems and conclusions.

2. Preliminaries. Let Σ be a fixed finite alphabet. Let $|y|$ denote the length of string $y \in \Sigma^*$, and let $||S||$ denote the cardinality of set $S \subseteq \Sigma^*$. Let $X \triangle Y$ denote $(X - Y) \bigcup (Y - X)$. For a set T, we define $T^{=n} = \{y \,|y \in T \text{ and } |y| = n\}$ and $T^{\leq n} = \{y \,|y \in T \text{ and } |y| \leq n\}$.

Let $\langle \cdot, \cdot \rangle_2$ denote a pairing function over finite strings (equivalently, over $\{0, 1, \cdots\}$ via the standard correspondence between strings and natural numbers), with the standard nice computability, invertibility, and other properties (e.g., for all $a, b \in \Sigma^*$ it holds that $|\langle a, b\rangle_2| = |a| + |b|$). Let $\langle y \rangle$ denote $\langle 1, y\rangle_2$, and, for every $k \geq 2$, let $\langle y_1, y_2, \cdots, y_k\rangle$ denote $\langle k, \langle y_1, \langle y_2, \langle \cdots, \langle y_{k-1}, y_k\rangle_2, \cdots\rangle_2\rangle_2\rangle_2\rangle_2$. This function is polynomial-time computable and polynomial-time invertible, and unambiguously codes a variable number of arguments.[3]

We adopt the standard notions of reducibility, as introduced by Ladner, Lynch, and Selman. (We make slight alterations in the definitions; these alterations do not effect the reductions defined.)

DEFINITION 2.1 [LLS75].

1. A *tt-condition* is a finite string of the form $\langle y_1, y_2, \cdots\rangle$, where each y_i is a member of Σ^*.
2. A *tt-condition generator* is a recursive mapping from Σ^* into the set of *tt-conditions*.
3. A *tt-condition evaluator* is a recursive mapping from $\{\langle x, \sigma_1, \cdots, \sigma_n\rangle \,|x \in \Sigma^*$ and $(\forall i)[\sigma_i \in \{0, 1\}]\}$ into $\{0, 1\}$. (We will use the convention of footnote 3 often on arguments of tt-condition evaluators.)
4. Let e be a tt-condition evaluator. A tt-condition $\langle y_1, \cdots, y_k\rangle$ is *e-satisfied on input x by* $B \subseteq \Sigma^*$ if and only if $e(x, \chi_B(y_1), \cdots, \chi_B(y_k)) = 1$.
5. Let g be a tt-condition generator and e be a tt-evaluator. Let $\lambda = (g, e)$. We will say that $\lambda^S(x)$ *accepts* if the tt-condition generated by $g(x)$ is e-satisfied on input x by S.
6. We say that $A \leq_{tt}^p B$ if there exist a polynomial-time computable generator g and a polynomial-time computable evaluator e such that, for all x, $x \in A \iff g(x)$ is e-satisfied on input x by B. If $A \leq_{tt}^p B$ we say that A is truth-table reducible to B in polynomial time.
7. We say that $A \leq_{f(n)-tt}^p B$ if $A \leq_{tt}^p B$ via a generator g and evaluator e such that on each input x, it holds that $g(x) \in \{\langle y_1, \cdots, y_k\rangle \,|y_i \in \Sigma^*$ and $k \leq f(|x|)\}$.

[3] For notational convenience, when speaking of functions f of more than one argument, we will freely write $f(y_1, \cdots)$ as a shorthand for $f(\langle y_1, \cdots\rangle)$.

For the special cases $\leq^p_{k-tt}$—that is, $f(n) = k$, $k \in \{1, 2, \cdots\}$—we will without loss of generality assume that the evaluator asks *exactly* k questions.

8. We say that $A \leq^p_{btt} B$ (A is polynomial-time bounded truth-table reducible to B) if $A \leq^p_{k-tt} B$ for some k.
9. We say that $A \leq^p_{ctt} B$ (A is polynomial-time conjunctive truth-table reducible to B) if $A \leq^p_{tt} B$ via a generator g and evaluator e such that the evaluator e has the property that for every x, $e(x, \sigma_1, \cdots, \sigma_k) = 1 \iff (\forall i : 1 \leq i \leq k)[\sigma_i = 1]$.
10. We say that $A \leq^p_{dtt} B$ (A is polynomial-time disjunctive truth-table reducible to B) if $A \leq^p_{tt} B$ via a generator g and evaluator e such that the evaluator e has the property that for every x, $e(x, \sigma_1, \cdots, \sigma_k) = 0 \iff (\forall i : 1 \leq i \leq k)[\sigma_i = 0]$.

In addition to the above reductions, we will also be concerned with nondeterministic reduction types. We defer the definitions of such reductions to §5.

Having defined the above types of reductions, we can now speak of the class of sets reducible or equivalent to a certain class of sets via a certain type of reduction. Such notions were first investigated in a systematic way by Book and Ko [BK88] and Tang and Book [TB]. We modify their nomenclature to allow a uniform notation for all reduction types.

DEFINITION 2.2.

1. Let $\mathcal{C}$ be a class of sets and let $\leq^t_r$ be a reducibility. We define

$$R^t_r(\mathcal{C}) = \{A \,|\, (\exists B)[B \in \mathcal{C} \text{ and } A \leq^t_r B]\} .$$

2. Let $\mathcal{C}$ be a class of sets and let $\leq^t_r$ be a reducibility. We define

$$E^t_r(\mathcal{C}) = \{A \,|\, (\exists B)[B \in \mathcal{C} \text{ and } A \leq^t_r B \text{ and } B \leq^t_r A]\} .$$

3. Many-one and 1-truth-table reductions. We first note that, if P = NP, then all sets many-one reducible to sparse sets are in fact many-one equivalent to sparse sets.

THEOREM 3.1. *If* P = NP *then* R^p_m(SPARSE)= E^p_m(SPARSE) $\bigcup \{\Sigma^*\}$.

Proof. Suppose $L \leq^p_m S$, S sparse, via many-one reduction $g(\cdot)$, and $L \neq \Sigma^*$. Define $S' = \{\langle 0^l, x\rangle \,|\, x \in S \text{ and } (\exists y)[y \in L \text{ and } |y| = l \text{ and } g(y) = x]\}$ (see [Mah82] for a similar "multiple-copy" approach). First, note that $L \leq^p_m S'$, as $y \in L \iff \langle 0^{|y|}, g(y)\rangle \in S'$. Second, note that $S' \leq^p_m L$ if P = NP. This is because, when asked whether $\langle 0^l, x\rangle \in S'$, we may use the fact that P = NP to determine whether there exists a y such that $|y| = l$ and $g(y) = x$. If not, reject $\langle 0^l, x\rangle$ by mapping onto an element out of L. If so, use the P = NP assumption to find one such y, call it y', and map to asking whether $y' \in L$. Finally, note that, immediately from the definition of S' and the fact that S is sparse, that S' is sparse. Thus, $L \in \mathrm{E}^p_m$(SPARSE). □

The proof of Theorem 3.1 can easily be modified to the case of 1-truth-table reductions. We need only change the definition of S' to $S' = \{\langle 0^l, x\rangle \mid$ either (1) $(\exists y)[y \in L$ and $|y| = l$ and the truth-table for input y accepts if and only if $x \in S]$, or (2) $(\exists y)[y \notin L$ and $|y| = l$ and the truth-table for input y accepts if and only if $x \notin S]\}$.

THEOREM 3.2. *If* P = NP *then* R^p_{1-tt}(SPARSE) = E^p_{1-tt}(SPARSE).

We say that a truth-table reduction, with truth-table condition generator g, is *honest* if there exists a polynomial $q(\cdot)$ such that whenever y is one of the query strings generated by $g(x)$, it holds that $q(|y|) \geq |x|$. Theorem 3.2 in fact gives honest equivalence.

Note that Theorem 3.1 does not establish that R^p_m(SPARSE) = E^p_m(SPARSE) $\bigcup \{\Sigma^*\}$ is equivalent to the claim that P=NP. We now note that analogous questions about tally sets are indeed equivalent to important open questions in complexity theory.

First, we present some definitions. A function f is *weakly invertible* if there is a polynomial-time computable function h such that $f(h(x)) = x$ for all $x \in range(f)$. Let E denote $\bigcup_{k\geq 0}$ DTIME$[2^{kn}]$, and let NE denote $\bigcup_{k\geq 0}$ NTIME$[2^{kn}]$.

It is shown in [AW90] that the following are equivalent:

1. Every NE predicate is E-solvable.
2. Every honest polynomial-time computable function $f : \Sigma^* \rightarrow 0^*$ is weakly invertible.
3. $\mathrm{E}^p_m(\mathrm{TALLY}) \bigcup \{\Sigma^*\} = \mathrm{E}^p_{1-tt}(\mathrm{TALLY})$.
4. $\mathrm{E}^p_m(\mathrm{TALLY}) \bigcup \{\Sigma^*\} = \mathrm{E}^p_{btt}(\mathrm{TALLY})$.

Condition 1 above is the natural "witness-finding" analog of the E = NE question. Impagliazzo and Tardos [IT89] have recently shown that there are relativized worlds in which Condition 1 fails to hold, yet E = NE. Their work provides a relativized refutation of a conjecture of Sewelson [Sew83], whose thesis forms the protasis of the[AW90], [IT89] research stream.

We note that the above equivalence can be extended to include classes of the form $\mathrm{R}^p_{btt}(\mathrm{TALLY})$, and, equivalently, $\mathrm{R}^p_m(\mathrm{TALLY})$. The following result, alluded to in [AH89], was observed independently by Fu Bin [Bin89].

THEOREM 3.3. *Every* NE *predicate is* E*-solvable if and only if*

$$\mathrm{R}^p_{btt}(\mathrm{TALLY}) = \mathrm{E}^p_m(\mathrm{TALLY}) \bigcup \{\Sigma^*\}.$$

Proof. Under the assumption that $\mathrm{R}^p_{btt}(\mathrm{TALLY}){=}\mathrm{E}^p_m(\mathrm{TALLY}) \bigcup \{\Sigma^*\}$, it follows immediately that $\mathrm{R}^p_{btt}(\mathrm{TALLY}) \subseteq \mathrm{E}^p_m(\mathrm{TALLY}) \bigcup \{\Sigma^*\} \subseteq \mathrm{E}^p_{btt}(\mathrm{TALLY}) \subseteq \mathrm{R}^p_{btt}(\mathrm{TALLY})$. Thus, $\mathrm{E}^p_m(\mathrm{TALLY}) \bigcup \{\Sigma^*\} = \mathrm{E}^p_{btt}(\mathrm{TALLY})$, and by the result of [AW90] mentioned above, it follows that every NE predicate is E-solvable.

Conversely, assume, via the above-mentioned equivalence of [AW90], that every honest polynomial-time computable function $f : \Sigma^* \rightarrow 0^*$ is weakly invertible, and let $L \leq^p_m T$, for some tally set T, via many-one reduction $g(\cdot)$. As in the proof of Theorem 3.1, define $T' = \{0^{\langle l, i+1\rangle} | 0^i \in T \text{ and } (\exists y)[y \in L \text{ and } |y| = l \text{ and } g(y) = 0^i]\}$. Then the function f defined by:

$$f(x) = \begin{cases} 0^{\langle |x|, i\rangle} & if\ g(x) = 0^i \\ 0^{\langle |x|, 0\rangle} & if\ g(x) \notin \{0^i\,|i \geq 0\} \end{cases}$$

is a many-one reduction from L to T'. Furthermore, under the assumption that f is weakly invertible (and assuming that $L \neq \Sigma^*$), it is easy to see that $T' \leq^p_m L$. Thus, under this assumption, $\mathrm{R}^p_m(\mathrm{TALLY}) = \mathrm{E}^p_m(\mathrm{TALLY}) \bigcup \{\Sigma^*\}$, and thus—via the fact that $\mathrm{R}^p_m(\mathrm{TALLY}) = \mathrm{R}^p_{btt}(\mathrm{TALLY})$ [BK88]—it holds that $\mathrm{R}^p_{btt}(\mathrm{TALLY}) = \mathrm{E}^p_m(\mathrm{TALLY}) \bigcup \{\Sigma^*\}$. □

4. Bounded-truth-table reductions.

4.1. A lower bound. Gavaldà and Watanabe have proven that for any unbounded polynomial-time computable function $f(n)$, $\mathrm{R}^p_{f(n)-tt}(\mathrm{SPARSE}) \not\subseteq \mathrm{E}^p_T(\mathrm{SPARSE})$ [GW]; their techniques do not seem to apply to the classes of sets reducible to sparse sets via $\leq^p_{k-tt}$ reductions, for constants k. However, Theorem 4.1, for the case of bounded truth-table reductions, a wide separation between reducibility and equivalence.

THEOREM 4.1. *Let* $h(n) = n^{o(1)}$. *Then* $\mathrm{R}^p_{2-tt}(\mathrm{SPARSE}) \not\subseteq \mathrm{E}^p_{h(n)-tt}(\mathrm{SPARSE})$.

Proof. For the purposes of this proof, we change our assumptions about the pairing function, and now assume that one of the properties of the pairing function $\langle\cdot\,,\cdot\rangle_2$ of §2

is that $(\forall x, y)[|\langle x, y\rangle_2| = 2|x| + |y|]$. Note that in this proof (and only in this proof) we will use both our standard pairing function $\langle \cdots \rangle$ and its constituent function $\langle \cdot , \cdot \rangle_2$.

Let us define an operator A such that, for any set S, $A(S) = \{\langle x, y\rangle_2 \mid |x| = |y|$ and x lexicographically precedes y and $(x \in S$ or $y \in S)\}$. Note that for any sparse set S, $A(S) \in \mathrm{R}^p_{2-tt}(\mathrm{SPARSE})$. We will construct a sparse set S so that $A(S) \notin \mathrm{E}^p_{h(n)-tt}(\mathrm{SPARSE})$.

In the following, for each $k \geq 1$, we use p_k to denote the polynomial $n^k + k$. Consider some enumeration $\{f_k\}_{k\geq 1}$ of $\leq^p_{h(n)-tt}$ reductions; without loss of generality, we may assume that, for all $k \geq 1$ and $x \in \Sigma^*$, the length of queries asked by $f_k(x)$ is bounded by $p_k(|x|)$. Let $C_{\langle i, j, l\rangle}$ denote the condition that, for each set W with census function bounded by p_l, either f_i is not a $\leq^p_{h(n)-tt}$ reduction from $A(S)$ to W, or f_j is not a $\leq^p_{h(n)-tt}$ reduction from W to $A(S)$.

Let us introduce some notations so that we may state the condition $C_{\langle i, j, l\rangle}$ more precisely. For any polynomial p, we say that set L is *p-sparse* if the census of L is bounded by p. For any set L, let $f_i^{-1}(L)$ denote the set $\{x \mid$ the truth-table condition of $f_i(x)$ evaluates to true when given L as the oracle$\}$. Then we can now restate $C_{\langle i, j, l\rangle}$ as the disjunction of the following two conditions:

I: $f_j^{-1}(A(S))$ is not p_l-sparse.

II: $A(S) \neq f_i^{-1}(f_j^{-1}(A(S)))$; that is, some v exists such that $v \in A(S) \iff v \notin f_i^{-1}(f_j^{-1}(A(S)))$.

We will build our set S in stages, where stage $\langle i, j, l\rangle$ will guarantee that $C_{\langle i, j, l\rangle}$ is satisfied. (Initially S is the empty set.) Note that this suffices to prove that S has the desired properties.

Stage $\langle i, j, l\rangle$:

Choose n large enough so that:

(i) interference with previous stages is avoided,

(ii) $(2^n/(2p_l \circ p_i(3n))^{h(3n)}) - h(3n)h(p_i(3n)) - 1 > 0$, and

(iii) $h(p_i(3n)) < n$.

(Note that such an n always exists since $h(n) = n^{o(1)}$.)

Case I: If there is a set $D \subseteq \Sigma^{=n}$, $||D|| \leq h(3n)h(p_i(3n)) + 1$, such that $f_j^{-1}(A(S \bigcup D))$ is not p_l-sparse, then set S to $S \bigcup D$.

Case II: If there is a set $D \subseteq \Sigma^{=n}$, $||D|| \leq h(3n)h(p_i(3n)) + 1$, such that $A(S \bigcup D) \neq f_i^{-1}(f_j^{-1}(A(S \bigcup D)))$, then set S to $S \bigcup D$.

(The construction fails if neither Case I nor Case II holds.)

If the above construction is completed, then the constructed set S clearly satisfies our purpose, that is, S is sparse and satisfies condition $C_{\langle i, j, l\rangle}$ for every $\langle i, j, l\rangle$. Thus, it remains only to show that the construction can be completed. That is, if Case I fails, then Case II must hold.

Consider any stage $\langle i, j, l\rangle$ and any sufficiently large n such that Case I does not hold. For such $\langle i, j, l\rangle$ and n, we show that Case II holds with some D. In the following discussion, let i, j, l, and n be fixed; let h denote $h(3n)$, and let h' denote $h(p_i(3n))$.

This paragraph gives an informal overview of the proof, in order to make the construction easier to understand. If Case I and Case II both fail, then there are sparse sets (call them W_1 and W_2) such that the following hold:

1. f_i is a $\leq^p_{h-tt}$ reduction from $A(S)$ to W_1 and f_j is a $\leq^p_{h-tt}$ reduction from W_1 to $A(S)$.

2. f_i is a $\leq^p_{h-tt}$ reduction from $A(S \bigcup \{0^n\})$ to W_2 and f_j is a $\leq^p_{h-tt}$ reduction from W_2 to $A(S \bigcup \{0^n\})$.

That is, only a small number of strings ($W_1 \cup W_2$) are sensitive to the presence or absence of 0^n in the set we are constructing. It follows that there is some string, w_1, that is queried by a large fraction of the strings in the set $\{\langle 0^n, y\rangle_2 \,|y \in \Sigma^*\}$ (recall that at this point, n is fixed). Thus w_1 may be thought of as being "influential" in some sense, and we can define Y_1 to be the (large) set of strings that are influenced by w_1. Let Z_1 be the (small) set of strings queried by the reduction f_j on input w_1. By setting membership for all the strings in Z_1, we completely determine membership for w_1, which means that there must be some string w_2 and some large subset $Y_2 \subseteq Y_1$ such that w_2 influences Y_2. We continue in this way until we arrive at a nonempty set of strings, each of which is influenced by (and thus queries) $x_1, x_2, \cdots, x_{h+1}$. But this is a contradiction, since no string makes more than h queries. This informal argument is made precise below.

We construct sets $D_1, \cdots, D_{k_0}, D_1^+, \cdots, D_{k_0}^+$ so that either D_{k_0} or $D_{k_0}^+$ satisfies Case II. The construction proceeds as follows:

Basis:

Set $Y_0 = \Sigma^{=n} - \{0^n\}$, $D_0 = \emptyset$, and $Z_0 = \emptyset$.

Definition of D_k and D_k^+ ($1 \leq k$):

Set $D_k = D_{k-1} \bigcup Z_{k-1}$, $D_k^+ = D_k \bigcup \{0^n\}$.

Set $A_k = A(S \bigcup D_k)$, $A_k^+ = A(S \bigcup D_k^+)$.

For each $w \in \Sigma^*$, set $Q_k(w) = \{y \in Y_{k-1} \,|f_i(\langle 0^n, y\rangle_2)$ queries $w\}$.

Set $C_k = \{w \,|Q_k(w) \neq \emptyset$ and $w \in f_j^{-1}(A_k) \triangle f_j^{-1}(A_k^+)\}$;

if C_k is empty, then terminate the construction.

Set w_k to be a string in C_k such that $||Q_k(w_k)|| \geq ||Y_{k-1}||/2p_l \circ p_i(3n)$;

if such w_k does not exist, then terminate the construction.

Set $Z_k = \{z \in \Sigma^{=n} - \{0^n\} \,|f_j(w_k)$ queries $\langle 0^n, z\rangle_2\}$.

Set $Y_k = Q_k(w_k) - Z_k$.

Now we show, in the following claims, that the construction terminates at some k_0, $1 \leq k_0 \leq h+1$. Note that the construction terminates either because C_{k_0} is empty or because no w_{k_0} exists. For each case, we prove that either $A_{k_0} \neq f_i^{-1}(f_j^{-1}(A_{k_0}))$ or $A_{k_0}^+ \neq f_i^{-1}(f_j^{-1}(A_{k_0}^+))$ occurs; that is, either D_{k_0} or $D_{k_0}^+$ satisfies Case II.

Before going further, let us explain the purpose of each of the sets in this construction. For each k, Y_k is a (large) set of strings that queries each of $\{w_1, w_2, \cdots, w_k\}$. Z_k is the set of strings queried by w_k, and $D_k = \bigcup_{s<k} Z_s$. (The strings $w_1, w_2, \cdots$ are chosen to be "influential," and the sets D_k are constructed so as to eliminate the influence of these strings.) D_k^+ is just $D_k \cup \{0^n\}$, and the sets A_k and A_k^+ are constructed from D_k and D_k^+ using the $A(\cdot)$ operator. C_k is the set of those strings that are sensitive to the difference between A_k and A_k^+, under reduction f_j.

Claim 1 states some properties that are immediate from the construction; its proof is omitted.

CLAIM 1.

(1) *For any k, $1 \leq k \leq h+1$, such that A_k and A_k^+ are defined:*

(a) $||D_k|| \leq ||D_k^+|| \leq (k-1)h' + 1$,

(b) $\{\langle 0^n, z\rangle_2 \,|z \in \bigcup_{1 \leq s < k} Z_s\} \subseteq A_k \subseteq A_k^+$,

(c) $||Y_{k-1}|| \geq (2^n/(2p_l \circ p_i(3n))^{k-1}) - (k-1)h' - 1 > 0$, and

(d) $A_k^+ - A_k \supseteq \{\langle 0^n, y\rangle_2 \,|y \in Y_{k-1}\} \neq \emptyset$.

(2) *Let k, $1 \leq k \leq h+1$, be any index such that Y_k is defined. For every $y \in Y_k$, $f_i(\langle 0^n, y\rangle_2)$ queries $w_1, \cdots, w_k$.*

The set C_k is the set of strings w such that (i) w is queried by $f_i(\langle 0^n, y\rangle_2)$ for some $y \in Y_{k-1}$, and (ii) $f_j(w)$ evaluates differently between oracle A_k and A_k^+. The following property of C_k is central to our construction.

CLAIM 2. *Let k, $1 \leq k \leq h+1$, be any index such that C_k is defined. For every s, $1 \leq s < k$, $w_s \notin C_k$.*

Proof. Note that every $\langle 0^n, z\rangle_2$ (except $\langle 0^n, 0^n\rangle_2$) that is queried by $f_j(w_s)$ is in A_k. Thus, the truth-table value of $f_j(w_s)$ does not vary by changing the oracle from A_k to A_k^+. □

CLAIM 3. *Suppose that the construction does not terminate at h. Then C_{h+1} is empty; thus, the construction terminates at most at $h+1$.*

Proof. It follows from Claim 1, part (2), that for every $y \in Y_h$, $f_i(\langle 0^n, y\rangle_2)$ queries $w_1, \cdots, w_h$. Since f_i is a $\leq^p_{h-tt}$ reduction, $\{w_1, \cdots, w_h\}$ is exactly the set of queries that are asked by $f_i(\langle 0^n, y\rangle_2)$ for some $y \in Y_h$. (Recall that $|\langle x, y\rangle_2| = 3n$ whenever $x \in \Sigma^{=n}$ and $y \in \Sigma^{=n}$.) Thus, $C_{h+1} \subseteq \{w_1, \cdots, w_h\}$. On the other hand, from Claim 2, none of $w_1, \cdots, w_h$ belongs to C_{h+1}. Therefore, $C_{h+1} = \emptyset$. □

CLAIM 4. *Suppose that the construction terminates at k_0. Then either $A_{k_0} \neq f_i^{-1}(f_j^{-1}(A_{k_0}))$ or $A_{k_0}^+ \neq f_i^{-1}(f_j^{-1}(A_{k_0}^+))$.*

Proof. First suppose that the construction terminates at k_0 because $C_{k_0} = \emptyset$. Since $k_0 \leq h+1$, it follows from Claim 1, part (1.d), that $\langle 0^n, y_0\rangle_2 \in A_{k_0}^+ - A_{k_0}$ for some $y_0 \in Y_{k_0-1}$. On the other hand, since $C_{k_0} = \emptyset$, the truth-table values of $f_i(\langle 0^n, y_0\rangle_2)$ relative to $f_j^{-1}(A_{k_0})$ and $f_j^{-1}(A_{k_0}^+)$ are the same. Hence, either $A_{k_0} \neq f_i^{-1}(f_j^{-1}(A_{k_0}))$ or $A_{k_0}^+ \neq f_i^{-1}(f_j^{-1}(A_{k_0}^+))$.

Next we show that if no w_{k_0} exists (and thus the construction terminates at k_0), then either $A_{k_0} \neq f_i^{-1}(f_j^{-1}(A_{k_0}))$ or $A_{k_0}^+ \neq f_i^{-1}(f_j^{-1}(A_{k_0}^+))$. We prove the contrapositive, i.e., for any k, $1 \leq k \leq h$, if $A_k = f_i^{-1}(f_j^{-1}(A_k))$ and $A_k^+ = f_i^{-1}(f_j^{-1}(A_k^+))$, then w_k certainly exists.

We show that $Y_{k-1} = \{z \mid (\exists w \in C_k)[z \in Q_k(w)]\}$ and $||C_k|| \leq 2p_l \circ p_i(3n)$, thereby proving that some $w_k \in C_k$ exists such that $||Q_k(w_k)|| \geq ||Y_{k-1}||/2p_l \circ p_i(3n)$.

Consider any $y \in Y_{k-1}$. Since $\langle 0^n, y\rangle_2$ is in $A_k \triangle A_k^+$ (from Claim 1, part (1.d)), and $A_k = f_i^{-1}(f_j^{-1}(A_k))$ and $A_k^+ = f_i^{-1}(f_j^{-1}(A_k^+))$ (from the assumption), $f_i(\langle 0^n, y\rangle_2)$ must query some w_y that is in $f_j^{-1}(A_k) \triangle f_j^{-1}(A_k^+)$. Recall that C_k is the set of strings in $f_j^{-1}(A_k) \triangle f_j^{-1}(A_k^+)$ that are queried by $f_i(\langle 0^n, y\rangle_2)$ for some $y \in Y_{k-1}$. Hence, w_y is in C_k. Thus, for each $y \in Y_{k-1}$, there is some $w_y \in C_k$ such that $f_i(\langle 0^n, y\rangle_2)$ queries w_y, i.e., $y \in Q_k(w_y)$; in other words, $Y_{k-1} = \{z \mid (\exists w \in C_k)[z \in Q_k(w)]\}$.

Recall that we are assuming that $f_j^{-1}(A(S \bigcup D))$ is p_l-sparse for any $D \subseteq \Sigma^{=n}$, $||D|| \leq hh' + 1$. Hence, both $f_j^{-1}(A_k)$ and $f_j^{-1}(A_k^+)$ are p_l-sparse; then clearly $C_k \subseteq f_j^{-1}(A_k) \triangle f_j^{-1}(A_k^+)$ is $2p_l$-sparse. Note that each $w \in C_k$ is queried by $f_i(\langle 0^n, y\rangle_2)$ for some $y \in \Sigma^{=n}$ and that the length of such a string is bounded by $p_i(|\langle 0^n, y\rangle_2|) \leq p_i(3n)$. (Recall that $|\langle x, y\rangle_2| = 3n$ for every x and $y \in \Sigma^{=n}$.) Thus, $||C_k|| \leq 2p_l(p_i(3n))$. □

This proves Theorem 4.1. □

The following is an immediate corollary.

COROLLARY 4.2. *For every $k \geq 2$, it holds that* $\mathrm{R}^p_{k-tt}(\mathrm{SPARSE}) \neq \mathrm{E}^p_{k-tt}(\mathrm{SPARSE})$.

4.2. Upper bounds. Corollary 4.2 establishes that, for all $k \geq 2$, $\mathrm{R}^p_{k-tt}(\mathrm{SPARSE}) \neq \mathrm{E}^p_{k-tt}(\mathrm{SPARSE})$. Gavaldà and Watanabe [GW] have proven that, for any unbounded polynomial-time computable function $f(n)$, $\mathrm{R}^p_{f(n)-tt}(\mathrm{SPARSE}) \not\subseteq \mathrm{E}^p_T(\mathrm{SPARSE})$. Both these results suggest that equivalence exacts a price; in order to achieve equivalence to sets reducible to sparse sets, one must use a more powerful type of reduction.

It is natural to seek the exact price that equivalence extracts. This section shows that, unless P $\neq$ NP, every set 2-truth-table reducible to a sparse set is truth-table equivalent to a sparse set. It follows that the result of Gavaldà and Watanabe cannot be extended to 2-truth-table reductions without providing a proof that P $\neq$ NP.

THEOREM 4.3. *If* P = NP *then* $\mathrm{R}^p_{2-tt}(\mathrm{SPARSE}) \subseteq \mathrm{E}^p_{tt}(\mathrm{SPARSE})$.

Proof. Let $L \leq^p_{2-tt} S$, S sparse, via truth-table generator g and evaluator e [LLS75]. Under our hypothesis that P = NP, we construct a sparse set $\widehat{S}$ such that $L \equiv^p_{tt} \widehat{S}$. Let $\{\tau_i \,|\, 1 \leq i \leq 16\}$ represent the sixteen truth-tables of arity two. Let $H_i = \{x \mid$ the truth table that $e(x, \cdot, \cdot)$ uses is table $\tau_i\}$. For each i, $1 \leq i \leq 16$, we will define a sparse set S_i and truth-table reductions $\lambda_i = (g_i, e_i)$ and $\gamma_i = (g'_i, e'_i)$ such that:

1. $(\forall i : 1 \leq i \leq 16)(\forall x \in H_i)[x \in L \iff \lambda_i^{S_i}(x)$ accepts] and
2. $(\forall i : 1 \leq i \leq 16)(\forall y)[y \in S_i \iff \gamma_i^{H_i \bigcap L}(y)$ accepts] and
3. $(\forall i : 1 \leq i \leq 16)(\forall y)[g'_i(y)$ queries only strings in H_i].[4]

Set $\widehat{S} = \{\langle i, j\rangle \,|\, j \in S_i \text{ and } 1 \leq i \leq 16\}$. By the above three conditions, $L \leq^p_{tt} \widehat{S}$ via the reduction that, on input x, determines which H_i contains x and uses λ_i modified so that each query z to S_i becomes a query $\langle i, z\rangle$ to $\widehat{S}$. Clearly, $\widehat{S} \leq^p_{tt} L$ via (g'', e''), where $g''(\langle i, z\rangle) = g'_i(z)$ and $e''(\langle i, z\rangle, \cdots) = e'_i(z, \cdots)$, for $1 \leq i \leq 16$, and as noted in footnote 4 for other i. Thus, $\widehat{S} \equiv^p_{tt} L$.

Figure 1 lists the sixteen truth-tables of size two. We proceed to define the sets $S_1, \cdots, S_{16}$.

Note that without loss of generality we make the following assumption.

ASSUMPTION 4.4. *g is length-increasing and* $g(x) = \langle b, c\rangle \Rightarrow |b| = |c|$.

This is simply because if $A \leq^p_{2-tt} B$, B sparse, via truth-table generator h, then $A \leq^p_{2-tt} B'$ via truth-table generator h', where $B' = \{\langle 0^l, y\rangle \,|\, y \in S\}$ and if $h(x)$ outputs $\langle q_1, q_2\rangle$ then $h'(x)$ outputs $\langle\langle 0^{|q_2|+|x|+1}, q_1\rangle, \langle 0^{|q_1|+|x|+1}, q_2\rangle\rangle$. Note that B' is sparse and h' maintains the properties asserted in Assumption 4.4 (recall that $|\langle a, b\rangle| = |a| + |b|$). We assume these properties throughout this proof.

Tables 1 and 16 are trivial; let $S_1 = S_{16} = \emptyset$. Tables 6, 8, 9, and 11 are 1-truth-table reductions; thus the construction of S_6, S_8, S_9, and S_{11} is essentially handled by Theorem 3.2. Similarly, Table 4 represents conjunctive 2-truth-table reductions and is essentially handled by the same result, since every set that conjunctive bounded truth-table reduces to a sparse set in fact many-one reduces to some sparse set.

Let us say that 2-truth-table a is the *complement* of 2-truth-table b if a and b differ on each possible response; for example, Tables 4 and 13 are complementary. Suppose we have proven that: (**) if A reduces to sparse set B via a 2-truth-table reduction that always uses Table τ, then $A \in \mathrm{E}^p_{tt}(\mathrm{SPARSE})$. It follows immediately that we have also proven: if A reduces to sparse set B via a 2-truth-table reduction that always uses Table $complement(\tau)$, then $A \in \mathrm{E}^p_{tt}(\mathrm{SPARSE})$. This is so because $\widehat{A}$ 2-truth-table reduces to $\widehat{B}$ via truth-table $complement(\tau)$ if and only if $\overline{\widehat{A}}$ 2-truth-table reduces to $\widehat{B}$ via truth-table τ. Thus, if we have established (**), we can conclude that $(\exists$ sparse set $C)[\overline{\widehat{A}} \equiv^p_{tt} C]$, and thus $\widehat{A} \equiv^p_{1-tt} \overline{\widehat{A}} \equiv^p_{tt} C$, so $\widehat{A} \equiv^p_{tt} C$. Thus it follows that the case of Table 13 follows immediately from that of Table 4. Below, we will use complementarity to reduce our work.

---

[4] We assume that each H_i is nonempty; the case where some H_i are empty can easily be dealt with by using vacuous truth-table reductions. For example, if $H_7 = \emptyset$, then set $S_7 = \emptyset$ and reduce S_7 to L via the truth-table evaluator that always rejects. Similarly, when resolving the membership in $\widehat{S}$ of elements of the form $\langle i, j\rangle$, with $i \notin \{1, 2, \cdots, 16\}$, we can also use a vacuous reduction.

Table Number	First Query Answered "no"		First Query Answered "yes"	
	2nd Ans. "no"	2nd Ans. "yes"	2nd Ans. "no"	2nd Ans. "yes"
1	0	0	0	0
2	1	0	0	0
3	0	0	1	0
4	0	0	0	1
5	0	1	0	0
6	1	0	1	0
7	1	0	0	1
8	1	1	0	0
9	0	0	1	1
10	0	1	1	0
11	0	1	0	1
12	1	0	1	1
13	1	1	1	0
14	1	1	0	1
15	0	1	1	1
16	1	1	1	1

FIG. 1. *Truth-tables of arity two.*

Consider the case of Table 15 (2-disjunctive reductions). Let $\widehat{S}_{15}$ represent all strings in S that are queried by some truth-table reduction from a member of H_{15}. Recalling Assumption 4.4, let polynomial $q(n)$ strictly upper-bound the number of elements in $\widehat{S}_{15}$ of length at most n. We will say that a string x is *busy* if there are more than $q(|x|)$ distinct strings w (each necessarily of the same length as x) that satisfy the condition:

there exists a string $\alpha_w \in H_{15} \bigcap L$ such that the (unordered) pair of strings queried by $g(\alpha_w)$ is $\{x, w\}$.

All busy strings are in $\widehat{S}_{15}$. However, there may also be strings in $\widehat{S}_{15}$ that are not busy. We now define $S_{15} = \{\langle 0^l, z\rangle \,|(\exists y, w)[|y| = l$ and the (unordered) pair of strings queried by $g(y)$ is $\{z, w\}$ and z is busy$]\} \bigcup \{\langle 0^l, y, z\rangle \,|(\exists w)[|w| = l$ and $w \in L$ and the (unordered) pair of strings queried by $g(w)$ is $\{y, z\}$ and neither y nor z is busy$]\}$.

Clearly, for strings in H_{15}, membership in L can be tested via $\leq^p_{3-tt}$ reduction to S_{15}, and clearly S_{15} is sparse.

CLAIM 1. *If* P = NP *then S_{15} truth-table reduces to L via a truth-table reduction that queries only members of H_{15}.*

Proof. There are two cases, corresponding to the two types of strings in S_{15}. In the first case, we are asked whether a string $\langle 0^l, z\rangle$ is in S_{15}. Use our assumption that P = NP to find[5] if possible (if not, then reject) more than $q(|z|)$ strings (not necessarily of length l) $\alpha_i \in H_{15}$, with each α_i mapping to $\{z, w_i\}$, with all the w_i's distinct, and use our P = NP assumption to find an appropriate y (of length l and in H_{15}). Then, via a

[5]Via binary search, in the standard fashion, using a test set such as $\{\langle z, \mathit{prefix}, \alpha_1, \cdots\rangle \mid$ there exists a string $\widehat{\alpha} \in H_{15}$ whose prefix is prefix and that differs from all the α_i and $g(\widehat{\alpha})$ yields the pair $\{z, p\}$ and this pair is not yielded by $g(\alpha_i)$ for any $i\}$.

truth-table query to L, check whether all the α_i are in L and accept if and only if all are.

In second case, we are asked whether a string $\langle 0^l, y, z\rangle$ is in S_{15}. Use our assumption that $\mathrm{P} = \mathrm{NP}$ to attempt to find more than $q(|z|)$ strings $\alpha_i \in H_{15}$, with each α_i mapping to $\{z, w_i\}$ with all the w_i's distinct. Also, use our assumption that $\mathrm{P} = \mathrm{NP}$ to attempt to find more than $q(|z|)$ strings $\beta_i \in H_{15}$, with each β_i mapping to $\{y, v_i\}$ with all the v_i's distinct. Finally, use our $\mathrm{P} = \mathrm{NP}$ assumption to find a string $w \in H_{15}$, of length l, such that $g(w)$ maps to the pair $\{y, z\}$. Now, we make a truth-table query to L, inquiring about the membership of w, the α_i's, and the β_i's. We accept if and only if (1) $w \in L$ and (2) either we failed to find the requisite number of α_i's or some of the α_i's found are not in L and (3) either we failed to find the requisite number of β_i's or some of the β_i's found are not in L. □

Note that, by the earlier complementation argument, solving Table 15 implicitly solves Table 2.

Consider now the case of Table 10 (exclusive or). Let $\widehat{S}_{10}$ represent all strings in S that are queried by some truth-table reduction from a member of H_{10}. Recalling Assumption 4.4, let polynomial $q(n)$ strictly upper-bound the number of elements in $\widehat{S}_{10}$ of length at most n. We will say that a string x is *heavy* if there are more than $q(|x|)$ distinct strings w (each necessarily of the same length as x) that satisfy the condition:

there exists a string[6] $\alpha_w \in H_{10} \bigcap L$ such that the (unordered) pair of strings queried by $g(\alpha_w)$ is $\{x, w\}$.

All heavy strings are in $\widehat{S}_{10}$. However, there may also be strings in $\widehat{S}_{10}$ that are not heavy. We now define $S_{10} = \{\langle 0^l, z\rangle \mid z \text{ is heavy}\} \bigcup \{\langle 0^l, w, z\rangle \mid w \text{ is heavy and } (\exists y)[|y| = l \text{ and } y \notin L \text{ and the (unordered) pair of strings queried by } g(y) \text{ is } \{w, z\}]\} \bigcup \{\langle 1^l, w, z\rangle \mid (\exists y)[|y| = l \text{ and } y \in L \text{ and the (unordered) pair of strings queried by } g(y) \text{ is } \{w, z\} \text{ and neither } w \text{ nor } z \text{ is heavy}]\}$.

Clearly, for strings in H_{10}, membership in L can be tested via $\leq^p_{5-tt}$ reduction to S_{10}, and clearly S_{10} is sparse.

CLAIM 2. *If* $\mathrm{P} = \mathrm{NP}$ *then* S_{10} *truth-table reduces to* L *via a truth-table reduction that queries only members of* H_{10}.

Proof. There are three cases, corresponding to the three types of strings in S_{10}.

Case 1. In the first case, we are asked whether a string $\langle 0^l, z\rangle$ is in S_{10}. Use our assumption that $\mathrm{P} = \mathrm{NP}$ to find (as before) as many α_i as possible (but no more than $2q(|z|) + 1$) such that $\alpha_i \in H_{10}$ and $g(\alpha_i)$ queries the (unordered) pair $\{z, w_i\}$ and $j \neq k \Rightarrow w_j \neq w_k$. If we have found $\leq q(|z|)$ such α_i's, then reject. Otherwise, make a truth-table query to L regarding the α_i's, and see if more than $q(|z|)$ of the α_i's are in L, and accept if and only if this is the case.

Note: the above strategy works since (1) if z is heavy, then there are no more than $q(|z|)$ values w_i such that some *nonmember* of $H_{10} - L$ maps to $\{z, w_i\}$ (as these w_i's must be in $\widehat{S}_{10}$), and (2) if z is not heavy, there can be at most $q(|z|)$ distinct values w_i such that some *member* of $L \bigcap H_{10}$ maps to $\{z, w_i\}$.

Case 2. In the second case, we are asked whether a string $\langle 0^l, w, z\rangle$ is in S_{10}. Check whether w is heavy as in Case 1. Also, use our $\mathrm{P} = \mathrm{NP}$ assumption to find a y as in the definition of S_{10}, and use L to check whether $y \notin L$. Accept if and only if an appropriate y was found and $y \notin L$ and w is heavy. (Note that all the above can be done via a single round of truth-table queries to L.)

Case 3. In the third case, we are asked whether a string $\langle 1^l, w, z\rangle$ is in S_{10}. Check

[6]Unlike the case of disjunctive reductions, in the exclusive-or case a string in $H_{10} - L$ may map to two strings in the sparse set, and we wish *not* to allow such cases to contribute towards heaviness.

that w is not heavy and that z is not heavy as in Case 1, except exchanging criteria (that is, if there are less than or equal to $q(|z|)$ values α_i then we find a string "not heavy," otherwise a string is "not heavy" if and only if no more than $q(|z|)$ of the α_i's are in L). Also, use our P = NP assumption to obtain y as in the definition of S_{10}, and use L to verify that $y \in L$. Accept if and only if an appropriate y exists and $y \in L$ and w is not heavy and z is not heavy. (Again, note that all the above can be done via a truth-table query to L.) □

Note that, by the earlier complementation argument, solving Table 10 implicitly solves Table 7.

Consider now the case of Table 3. Let $\widehat{S}_3$ represent all strings in S that are queried by some truth-table reduction from a member of H_3. Recalling Assumption 4.4, let polynomial $q(n)$ strictly upper-bound the number of elements in $\widehat{S}_3$ of length at most n. We will say that a string x is *top-heavy* if there are more than $q(|x|)$ distinct strings w (each necessarily of the same length as x) that satisfy the condition:

there exists a string $\alpha_w \in H_3 \bigcap L$ such that the (ordered) pair of strings queried by $g(\alpha_w)$ is (x, w).

All top-heavy strings are in $\widehat{S}_3$. However, there may also be strings in $\widehat{S}_3$ that are not top-heavy. We now define $S_3 = \{\langle 0^l, z\rangle \mid z \text{ is top-heavy}\} \bigcup \{\langle 0^l, z', z''\rangle \mid z' \text{ is not top-heavy and } (\exists y)[|y| = l \text{ and } y \in L \bigcap H_3 \text{ and } g(y) = \langle z', z''\rangle]\} \bigcup \{\langle 1^l, z, z'\rangle \mid z \text{ is top-heavy and } (\exists w)[|w| = l \text{ and } w \in H_3 - L \text{ and } g(w) = \langle z, z'\rangle]\}$.

Clearly, for strings in H_3, membership in L can be tested via $\leq^p_{3-tt}$ reduction to S_3, and clearly S_3 is sparse.

CLAIM 3. *If* P = NP *then* S_3 *truth-table reduces to* L *via a truth-table reduction that queries only members of* H_3.

Proof. There are three cases, corresponding to the three types of strings in S_3.

Case 1. In the first case, we are asked whether a string $\langle 0^l, z\rangle$ is in S_3. Use our assumption that P = NP to find (as before) as many α_i as possible (but no more than $2q(|z|) + 1$) such that $\alpha_i \in H_3$ and $g(\alpha_i) = \langle z, w_i\rangle$ and $j \neq k \Rightarrow w_j \neq w_k$. If we have found more than $q(|z|)$ such α_i's that are in L, then accept; otherwise reject.

Note: the above strategy works since if z is top-heavy, then there are no more than $q(|z|)$ values α_i as above such that $\alpha_i \in H_3 - L$.

Case 2. In the second case, we are asked whether a string $\langle 0^l, z', z''\rangle$ is in S_3. Check whether z' is *not* top-heavy as in Case 1, except flipping our notions of acceptance and rejection. Also, use our P = NP assumption to find y as in the definition of S_3 (reject if there is no such y). Accept if and only if y is in L and z' is not top-heavy.

Case 3. In the third case, we are asked whether a string $\langle 1^l, z, z'\rangle$ is in S_3. Check whether z is top-heavy as in Case 1. Also, use our P = NP assumption to find w as in the definition of S_3 (reject if there is no such w). Accept if and only if w is not in L and z is top-heavy. □

Note that, by the earlier complementation argument, and by symmetry, and by both complementation and symmetry, solving Table 3 implicitly solves Tables 14, 5, and 12. This proves Theorem 4.3. □

In fact, a careful inspection of the proof of Theorem 4.3 reveals that various stronger statements than Theorem 4.3 have been implicitly proven. These improvements show, among other things, that Theorem 4.1 cannot be improved. The power of unbounded-truth-table reductions, and the strength of the P = NP assumption, are both used only in one direction. Thus we have Theorem 4.5.

THEOREM 4.5. *Let* $\epsilon > 0$. *If* $L \in \mathrm{R}^p_{2-tt}(\mathrm{SPARSE})$ *then there exists a sparse set* S' *such that:*

- $L \leq^{p}_{5-tt} S'$, *and*
- *if* P = NP *then* $S' \leq^{p}_{n^{\epsilon}-tt} L$.

Proof. The proof of Theorem 4.3 provides a set $\widehat{S}$ such that $L \leq^{p}_{5-tt} \widehat{S}$, and if P = NP then $\widehat{S} \leq^{p}_{n^k-tt} L$ for some k. Now let $S' = \{x10^{|x|^l} : x \in \widehat{S}\}$ for some l such that $k/l < \epsilon$; it is immediate that $L \leq^{p}_{5-tt} S'$, and if P = NP then $S' \leq^{p}_{n^{\epsilon}-tt} L$. □

Of course, the assumption that P = NP is a very strong one. However, though the P = NP assumption gives polynomial-time computations access to the full power of the polynomial-hierarchy, in fact the above proofs used the P = NP assumption only to give polynomial-time computations access to the power of NP (and in particular, the power to find sets of inverses of honest polynomial-time many-one functions). Thus the above proof in fact proves Theorem 4.6 below, whose oracle access mechanism is exactly that used in defining the extended-low-two sets [BBS86]—a mechanism that also appears in other applications [HH90]. Of particular note is that the set L is—as in Theorem 4.3 but unlike Theorem 4.7—queried only polynomially often.

THEOREM 4.6. *If* $L \in \mathrm{R}^{p}_{2-tt}(\mathrm{SPARSE})$ *then there exists a sparse set* S' *such that:*

- $L \leq^{p}_{5-tt} S'$, *and*
- $S' \in \mathrm{P}^{\mathrm{NP} \oplus L}$.

The above results are all conditioned upon the assumption that P = NP or the essentially equivalent use of an NP oracle. In fact, we can outright eliminate such assumptions, at the cost of acquiescing to relatively powerful reductions that are allowed to access the set L far more than polynomially often. Thus, the following theorem neither implies nor is implied by Theorem 4.6.

THEOREM 4.7. *If* $L \in \mathrm{R}^{p}_{2-tt}(\mathrm{SPARSE})$ *then there exists a sparse set* S' *such that:*

- $L \leq^{p}_{5-tt} S'$, *and*
- $S' \in \mathrm{DP}^{L}$.

Here, DP, difference polynomial time, is the class of sets—first studied by Papadimitriou and Yannakakis [PY84]—that can be represented as the difference of two NP sets; DP sets are crucial to the normal-form structure of the boolean hierarchy [CGH+88] and appear naturally in many settings [CM87]. Informally, we may describe Theorem 4.7 as stating that all sets 2-truth-table reducible to sparse sets are DP-equivalent to sparse sets. We omit the proof, as it is based on a detailed analysis similar to that of Theorem 4.3.

Finally, we note that all the theorems of this section yield not only equivalence but indeed honest equivalence.

5. On the power of conjunctive and disjunctive reductions. In this section, we will show several inclusions among classes of sets that are reducible to sparse sets. We first show the following lemma.

LEMMA 5.1. $\mathrm{R}^{p}_{1-tt}(\mathrm{SPARSE}) \subseteq \mathrm{R}^{p}_{dtt}(\mathrm{SPARSE})$.

Proof. Let L be a set that is $\leq^{p}_{1-tt}$ reducible to a sparse set S. We will show that $L \leq^{p}_{dtt} U$ for some sparse set U. To prove this, we need to define some notation. For string x and $n \geq 1$, we will use x_n to denote the nth symbol in x. For two strings x and y and a set A, xAy denotes the set $\{xwy \mid w \in A\}$. Let # be a special symbol not in Σ. For two sets A and B, $A \oplus B$ denotes the set $0A \bigcup 1B$. Since S is sparse, there exists a polynomial p_0 such that for every $n \geq 0$ it holds that $||S^{\leq n}|| \leq p_0(n)$.

Let $T = \{0z0 \mid z \in S\} \bigcup 01^*$. It is not hard to see that for every $x \in \Sigma^*$, $x \in S \iff 0x0 \in T$ and for every $n \geq 1$, $T^{=n} \neq \emptyset$. Moreover, for every $n \geq 0$,

$$||T^{\leq n}|| \leq ||S^{\leq n-2}|| + n \leq ||S^{\leq n}|| + n \leq p_0(n) + n.$$

Therefore, T is sparse.

Now define U to be the set of strings of the form $\#^n u\#b$ such that:
(1) $u \in \Sigma^{\leq n}$ and $b \in \Sigma$,
(2) $ub\Sigma^* \bigcap T^{=n} = \emptyset$; that is, ub is not a prefix of any string in $T^{=n}$, and
(3) $u\Sigma^* \bigcap T^{=n} \neq \emptyset$; that is, u is a prefix of some string in $T^{=n}$.
Then, for every $n \geq 0$:

$$\begin{aligned} ||U^{\leq n}|| &\leq ||\{\#^m u\#b \in U \,|1 \leq m \leq n\}|| \\ &\leq ||\{\#^m u\#b \,|\, 1 \leq m \leq n \text{ and } b \in \Sigma \text{ and } u\Sigma^* \bigcap T^{=m} \neq \emptyset\}|| \\ &\leq 2n||\{w \,|w \text{ is a prefix of some string in } T^{\leq n}\}|| \\ &\leq 2n^2||T^{\leq n}|| \\ &\leq 2n^2(p_0(n)+n). \end{aligned}$$

Therefore, U is sparse.

We now establish the following claim.

CLAIM 1. *Let $x \in \Sigma^*$ and $y = 0x0$. Then, $y \notin T$ if and only if $\{\#^{|y|}y_1 \cdots y_k\#y_{k+1} \mid 1 \leq k \leq |y|-1\} \bigcap U \neq \emptyset$ and $y \in T$ if and only if $\#^{|y|}y\#0 \in U$.*

Proof. Let x be any string and let $y = 0x0$. Furthermore, let n denote $|y|$ $(= |x|+2)$. First suppose that $y \in T$. Since $|y0| > n$ and $y \in T^{=n}$, $\#^n y\#0 \in U$. Furthermore, for every $i, 1 \leq i \leq n$, $y_1 \cdots y_i$ is clearly a prefix of y. Thus, for every $k, 1 \leq k < n$, $\#^n y_1 \cdots y_k\#y_{k+1}$ is not in U.

On the other hand, suppose that $y \notin T$. Clearly, $\#^n y\#0 \notin U$ because $y \notin T^{=n}$. Furthermore, since for every $n \geq 1$ it holds that $T^{=n} \neq \emptyset$ and $T^{=n} \subseteq 0\Sigma^*$, there exists a unique $m, 1 \leq m < n-1$, such that $y_1 \cdots y_m$ is a prefix of some string in $T^{=n}$ but $y_1 \cdots y_{m+1}$ is not a prefix of any string in $T^{=n}$. So, clearly, $\#^n y_1 \cdots y_m\#y_{m+1} \in U$.

From the above considerations, $y \in T$ if and only if $\#^n y\#0 \in U$ and $y \notin T$ if and only if for some $k, 1 \leq k < n$, $\#^n y_1 \cdots y_k\#y_{k+1} \in U$. This proves the claim. □

Since $x \in S$ if and only if $y = 0x0 \in T$, from Claim 1, we have:

$$\begin{aligned} x \in S &\iff \#^{|y|}y\#0 \in U \quad \text{and} \\ x \notin S &\iff (\exists k : 1 \leq k \leq |y|-1)[\#^{|y|}y_1 \cdots y_k\#y_{k+1} \in U]. \end{aligned}$$

Therefore, $S \oplus \bar{S} \leq_{dtt}^p U$. Since $L \leq_{1-tt}^p S$ implies that $L \leq_m^p S \oplus \bar{S}$, $L \leq_{dtt}^p U$ and this proves the lemma. □

From Lemma 5.1, we obtain the following theorem.

THEOREM 5.2. $\mathrm{R}_{btt}^p(\mathrm{SPARSE}) \subseteq \mathrm{R}_{dtt}^p(\mathrm{SPARSE})$.

Proof. Let L be a set $\leq_{k-tt}^p$ reducible to a sparse set S for some $k \geq 0$ via a polynomial-time computable function f. To establish the theorem, we have only to show that there is another sparse set A to which L is $\leq_{dtt}^p$ reducible. Since f witnesses that $L \leq_{k-tt}^p S$, without loss of generality (see [LLS75]), we may assume the following: For every $x \in \Sigma^*$,

(a) $f(x)$ is of the form $b_{11} \cdots b_{1k}\$ \cdots \$b_{m1} \cdots b_{mk}\$w_1\$ \cdots \$w_k$, where (1) \$ is a new symbol not in $\{0, 1, \#\}$, (2) for every $i, 1 \leq i \leq m$ and $j, 1 \leq j \leq k$, $b_{ij} \in \{0, 1\}$, and (3) for every $i, 1 \leq i \leq m$, $w_i \in \Sigma^*$, and

(b) $x \in L$ if and only if $(\exists i : 1 \leq i \leq m)(\forall j : 1 \leq j \leq k)[\chi_S(w_j) = \text{true} \iff b_{ij} = 0]$, where χ_S is the characteristic function of S; that is, for every w, $\chi_S(w) = $ true if $w \in S$ and false otherwise.

Since $S \oplus \bar{S}$ is $\{0s \,|s \in S\} \bigcup \{1s \,|s \in \bar{S}\}$, it is not hard to see that the condition (b) is equivalent to the following:

(b1) $x \in L$ if and only if $(\exists i : 1 \le i \le m)(\forall j : 1 \le j \le k)[b_{ij}w_j \in S \oplus \bar{S}]$.

Now recall that we showed in Lemma 5.1 that there is a sparse set U to which $S \oplus \bar{S}$ is $\le^p_{dtt}$ reducible. Let g be a $\le^p_{dtt}$ reduction from $S \oplus \bar{S}$ to U. Then, without loss of generality, we may assume that for every $y \in \Sigma^*$,

(c) $g(y)$ is of the form $z_1\$\cdots\$z_m$, where $m = p(|y|)$ for some polynomial p and

(d) $y \in S \oplus \bar{S}$ if and only if $\{z_1, \cdots, z_m\} \bigcap U \ne \emptyset$.

For each y, let $\sigma(y)$ denote the set of all strings $\{z_1, \cdots, z_m\}$ that g outputs upon input y. Then, the condition (b1) is equivalent to the following:

(b2) $x \in L$ if and only if $(\exists i : 1 \le i \le m)(\forall j : 1 \le j \le k)[\sigma(b_{ij}w_j) \bigcap U \ne \emptyset]$.

For each $i, 1 \le i \le m$, let $H(i)$ denote the set:

$$\{u_1\$\cdots\$u_k \,|(\forall j : 1 \le j \le k)[u_j \in \sigma(b_{ij}w_j)]\}.$$

Also, define:

$$A = \{u_1\$\cdots\$u_k \,|(\forall j : 1 \le j \le k)[u_j \in U]\}.$$

Then, it is not hard to see that the condition (b2) is equivalent to:

(b3) $x \in L$ if and only if $(\exists i : 1 \le i \le m)(\exists v \in H(i))[v \in A]$.

Since f and g are polynomial-time computable and k is a constant, there is a polynomial q such that $||\{z \,|(\exists i : 1 \le i \le m)[z \in H(i)]\}|| \le q(|x|)$.

Furthermore, it is easy to see that the set $\{z \,|(\exists i : 1 \le i \le m)[z \in H(i)]\}$ is polynomial-time computable in $|x|$. So, let h be a function that computes $v_1\$\cdots\$v_n$ so that $v_1, \cdots, v_n$ is an enumeration of all strings in $H(i)$ for some $i, 1 \le i \le m$. h is polynomial-time computable. $x \in L$ if and only if $\{v_1, \cdots, v_n\} \bigcap A \ne \emptyset$. Thus, h witnesses $L \le^p_{dtt} A$. Finally, since U is sparse and k is a constant, clearly A is sparse. Therefore, $L \in R^p_{dtt}$(SPARSE), and this proves the theorem. □

Next we consider the classes of sets that are reducible to sparse sets via polynomial-time nondeterministic Turing machines. The following definitions are due to Ladner, Lynch, and Selman.

DEFINITION 5.3 [LLS75].

(1) A set A is polynomial-time nondeterministic many-one reducible to a set B (denoted $A \le^{NP}_m B$) if there exists a polynomial-time nondeterministic Turing machine M such that for every $x \in \Sigma^*$,
   (1A) for each computation path of M on x, M outputs some string, and
   (1B) $x \in A$ if and only if there exists some string $y \in B$ that M outputs for some computation path on input x.

(2) A set A is polynomial-time nondeterministic Turing reducible to a set B (denoted $A \le^{NP}_T B$) if there exists a polynomial-time nondeterministic oracle Turing machine M such that for every $x \in \Sigma^*$, $x \in A$ if and only if there exists an accepting computation path of M on x relative to B.

(3) A set A is polynomial-time nondeterministic bounded truth-table reducible to a set B (denoted $A \le^{NP}_{btt} B$) if there exist $k \ge 0$ and a polynomial-time nondeterministic Turing machine M such that for every $x \in \Sigma^*$,
   (3A) for each computation path of M on x, M outputs a string of the form $(\alpha, y_1, \cdots, y_k)$, where α is a k-truth-table and $y_1, \cdots, y_k \in \Sigma^*$, and
   (3B) $x \in A$ if and only if there exists some output $(\alpha, y_1, \cdots, y_k)$ of M on x for some computation path such that $\alpha(\chi_B(y_1), \cdots, \chi_B(y_k)) =$ true, where χ_B is the characteristic function of B.

(4) A set A is polynomial-time nondeterministic truth-table reducible to a set B, denoted by $A \leq_{tt}^{NP} B$, if there exists a polynomial-time nondeterministic Turing machine M and a polynomial-time computable truth-table evaluator such that $x \in A$ if and only if M, on input x, computes on some computation path a tt-condition y that is e-satisfied by B.

(5) A set A is polynomial-time nondeterministic conjunctive truth-table reducible to a set B, denoted by $A \leq_{ctt}^{NP} B$, if there exists a polynomial-time nondeterministic Turing machine M such that for every $x \in \Sigma^*$,

(5A) for each computation path of M on x, M outputs a string of the form $(y_1, \cdots, y_k)$, where $y_1, \cdots, y_k \in \Sigma^*$, and

(5B) $x \in A$ if and only if there exists some output $(y_1, \cdots, y_k)$ of M on x for some computation path such that $\{y_1, \cdots, y_k\} \subseteq B$.

(6) A set A is polynomial-time nondeterministic disjunctive truth-table reducible to a set B, denoted by $A \leq_{dtt}^{NP} B$, if there exists a polynomial-time nondeterministic Turing machine M such that for every $x \in \Sigma^*$,

(6A) for each computation path of M on x, M outputs a string of the form $(y_1, \cdots, y_k)$, where $y_1, \cdots, y_k \in \Sigma^*$, and

(6B) $x \in A$ if and only if there exists some output $(y_1, \cdots, y_k)$ of M on x for some computation path such that $\{y_1, \cdots, y_k\} \bigcap B \neq \emptyset$.

DEFINITION 5.4. R_m^{NP}(SPARSE)(R_T^{NP}(SPARSE), R_{btt}^{NP}(SPARSE), R_{tt}^{NP}(SPARSE), R_{ctt}^{NP}(SPARSE), R_{dtt}^{NP}(SPARSE)) denotes the class of sets that are $\leq_m^{NP}$ (respectively, $\leq_T^{NP}$, $\leq_{btt}^{NP}$, $\leq_{tt}^{NP}$, $\leq_{ctt}^{NP}$, $\leq_{dtt}^{NP}$) reducible to some sparse set.

We may also use Lemma 5.1 to obtain the following theorem. It is important to emphasize that the results of this section depend crucially on the fact that we are reducing to the class of *sparse* sets. In particular, the following theorem should be contrasted with the fact that there are classes $\mathcal{C}$, and indeed single sets, such that $\mathrm{R}_{btt}^{NP}\mathcal{C}$ and $\mathrm{R}_m^{NP}\mathcal{C}$ differ [LLS75].

THEOREM 5.5. R_m^{NP}(SPARSE) $=$ R_{btt}^{NP}(SPARSE)

Proof. To prove this, we will show that R_{btt}^{NP}(SPARSE) $\subseteq$ R_{dtt}^{NP}(SPARSE). Ladner, Lynch, and Selman [LLS75, Thm. 4.1, Part (iii)] have shown that for every pair of sets A and B, it holds that $A \leq_m^{NP} B$ if and only if $A \leq_{dtt}^{NP} B$. It follows immediately that R_{dtt}^{NP}(SPARSE) $\subseteq$ R_m^{NP}(SPARSE). Thus, it suffices to show that R_{btt}^{NP}(SPARSE) $\subseteq$ R_{dtt}^{NP}(SPARSE).

Let L be a set that, for some k, is $\leq_{k-tt}^{NP}$ reducible to a sparse set S via polynomial-time nondeterministic Turing machine M. Without loss of generality, we may assume that there is a polynomial p such that for every $x \in \Sigma^*$, each computation path of M on x has length exactly $p(|x|)$. Define $A = \{x\#y \,|\, y \in \Sigma^{=p(|x|)}$ and $M(x)$ on computation path y has output of the form $(\alpha, y_1, \cdots, y_k)$ such that $\alpha(\chi_S(y_1), \cdots, \chi_S(y_k)) = \text{true}\}$. It is not hard to see that $A \leq_{k-tt}^{p} S$, and for every $x \in \Sigma^*$, $x \in L$ if and only if for some $y \in \Sigma^{=p(|x|)}$ it holds that $x\#y \in A$. Since $A \leq_{k-tt}^{p} S$, from Theorem 5.2, there exist a sparse set S' and a polynomial-time computable function f such that $A \leq_{dtt}^{p} S'$ is witnessed by f. Consider a machine N that, on input $x \in \Sigma^*$, nondeterministically guesses $y \in \Sigma^{=p(|x|)}$ and outputs $f(x\#y)$. Clearly, the machine N witnesses $L \leq_{dtt}^{NP} S'$. □

Next we prove the following theorem.

THEOREM 5.6. R_{ctt}^{NP}(SPARSE) $=$ R_T^{NP}(SPARSE).

Proof. Let L be a set in R_T^{NP}(SPARSE). Thus, there exists a polynomial-time nondeterministic oracle Turing machine and a sparse set S such that for every x, $x \in L$ if and only if M on input x relative to S accepts. Here, without loss of generality, we may

assume the following: There exist two polynomials p and q such that for every x,

(1) M on input x has exactly $p(|x|)$ nondeterministic steps for each computation path, and

(2) for every computation path and for every oracle set X, M on x queries the oracle set exactly $q(|x|)$ times.

We will encode each computation path of M on input x as a string of length $p(|x|)$.

Moreover, since S is sparse, there exist a polynomial-time computable function f and a sparse set U such that $S \oplus \bar{S} \leq_{dtt}^{p} U$ via f.

Now consider the following nondeterministic Turing machine M':

1. On input x, M' nondeterministically guesses a string $w \in \Sigma^{=p(|x|)}$ and $b_1, \cdots, b_{q(|x|)} \in \{0, 1\}$. M' simulates the computation of M on input x for the computation path w in the following way: whenever the ith query y_i is made, instead of querying to the oracle M' regards the answer to the query as YES if $b_i = 0$ and NO otherwise, and M' stores the query string on its tape. If the simulation of M on x terminates at an accepting state, then M' proceeds to the next step. Otherwise, M' outputs a fixed string not in U and halts.
2. For each $i, 1 \leq i \leq q(|x|)$, M' looks up the table of query strings computed in the previous step, computes $f(b_i y_i)$, and nondeterministically picks a string z_i in the output of f.
3. Finally, M' outputs $z_1\$ \cdots \$z_k$ and halts.

From the above description, as in the proof of the previous theorem, it is not hard to see that (1) M' runs in time polynomial in $|x|$ and (2) $x \in L$ if and only if M' on x outputs some $(z_1, \cdots, z_{q(|x|)})$ such that $\{z_1, \cdots, z_{q(|x|)}\} \subseteq U$. Therefore, M' witnesses that $L \in \mathrm{R}_{ctt}^{NP}$(SPARSE), thus proving the theorem. □

6. Conclusions and open problems. This paper addressed the question of whether reducibility to sparse sets is a broader notion than equivalence to sparse sets. For the many-one and 1-truth-table cases, we showed that differentiating reducibility from equivalence would yield a proof that P $\neq$ NP. In contrast, for the k-truth-table case, $k \geq 2$, reducibility *is* a provably broader notion than equivalence.

Nonetheless, there are limits on how much broader it can be. Gavaldà and Watanabe have proven that for every nice unbounded function f, some sets $f(n)$-truth-table reducible to sparse sets are not Turing equivalent (or even strong-nondeterministic equivalent) to any sparse set. However, we showed that their result cannot be extended to the 2-truth-table case without yielding a proof that P $\neq$ NP. In particular, if P = NP then all sets 2-truth-table reducible to sparse sets are truth-table equivalent to sparse sets.

Finally, we addressed the power of disjunctive and conjunctive reductions to sparse sets. Refuting a conjecture of Ko [Ko89], we proved that all sets bounded truth-table reducible to sparse sets are indeed disjunctive truth-table reducible to sparse sets. Relatedly, for nondeterministic reductions to sparse sets, we proved that bounded truth-table reductions are no stronger than many-one reductions, and that Turing reductions are no stronger than conjunctive truth-table reductions.

A number of questions remain open. Regarding §5, though we refuted Ko's conjecture about disjunctive reductions, Ko's other conjectures have as yet been neither proven nor refuted. Regarding §4.2, can our proof be generalized from the 2-truth-table case to the bounded truth-table case?

A particularly interesting issue is that, even in the wake of the Gavaldà and Watanabe's study of the case of Turing reductions, many of the same questions remain open for the Turing case. Gavaldà and Watanabe [GW] show that not all sets Turing reducible to sparse sets are even strong-nondeterministic equivalent to sparse sets. This is essentially

an NP $\bigcap$ coNP lower bound on the strength of the reduction needed to achieve equivalence. A moment's thought reveals—via [Sch86b, Lemma 5.6]—an upper bound of Σ_2^p; that is, every set Turing reducible to a sparse set is $\equiv_T^{\Sigma_2^p}$ to some other sparse set.[7] However, the exact location of the optimal strength of reduction needed to achieve equivalence has not yet been pinpointed more accurately than the range (NP $\bigcap$ coNP, Σ_2^p].

Acknowledgments. We are grateful to Ronald Book for making our collaboration possible, to Russell Impagliazzo, Sanjay Jain, Robert Szelepcsényi, and Jozef Vyskoč for helpful comments and conversations, and to two anonymous referees for helpful suggestions. We thank the Tokyo Institute of Technology for hosting a workshop on computational complexity, in August 1990, at which this work was done in part. We are particularly grateful to Ricard Gavaldà and an anonymous referee for pointing out an error in an earlier version of this paper.

REFERENCES

[AH] E. ALLENDER AND L. HEMACHANDRA, *Lower bounds for the low hierarchy*, J. ACM, 39 (1992), pp. 234–250.

[AH89] ———, *Lower bounds for the low hierarchy*, in Proc. 16th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, 372, Springer-Verlag, Berlin, New York, 1989, pp. 31–45.

[AW90] E. ALLENDER AND O. WATANABE, *Kolmogorov complexity and the degrees of tally sets*, Inform. Comput., 86 (1990), pp. 160–178.

[BB86] J. BALCÁZAR AND R. BOOK, *Sets with small generalized Kolmogorov complexity*, Acta Inform., 23 (1986), pp. 679–688.

[BBS86] J. BALCÁZAR, R. BOOK, AND U. SCHÖNING, *Sparse sets, lowness and highness*, SIAM J. Comput., 15 (1986), pp. 739–746.

[BDG88] J. BALCÁZAR, J. DÍAZ, AND J. GABARRÓ, *Structural Complexity I*, EATCS Monographs in Theoretical Computer Science, Springer-Verlag, Berlin, New York, 1988.

[BH77] L. BERMAN AND J. HARTMANIS, *On isomorphisms and density of NP and other complete sets*, SIAM J. Comput., 6 (1977), pp. 305–322.

[Bin89] F. BIN, Sept. 1989, personal communication.

[BK88] R. BOOK AND K. KO, *On sets truth-table reducible to sparse sets*, SIAM J. Comput., 17 (1988), pp. 903–919.

[CGH+88] J. CAI, T. GUNDERMANN, J. HARTMANIS, L. HEMACHANDRA, V. SEWELSON, K. WAGNER, AND G. WECHSUNG, *The boolean hierarchy I: Structural properties*, SIAM J. Comput., 17 (1988), pp. 1232–1252.

[CGH+89] ———, *The boolean hierarchy II: Applications*, SIAM J. Comput., 18 (1989), pp. 95–111.

[CM87] J. CAI AND G. MEYER, *Graph minimal uncolorability is D^P-complete*, SIAM J. Comput., 16 (1987), pp. 259–277.

[GW] R. GAVALDÀ AND O. WATANABE, *On the computational complexity of small descriptions*, in Proc. 6th Structure in Complexity Theory Conference, IEEE Computer Society Press, June/July 1991, pp. 89–101.

[HH90] J. HARTMANIS AND L. HEMACHANDRA, *Robust machines accept easy sets*, Theoret. Comput. Sci., 74 (1990), pp. 217–226.

[HM80] J. HARTMANIS AND S. MAHANEY, *An essay about research on sparse NP complete sets*, in Proc. 9th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, 88, Springer-Verlag, Berlin, New York, 1980, pp. 40–57.

[IM89] N. IMMERMAN AND S. MAHANEY, *Relativizing relativized computations*, Theoret. Comput. Sci., 68 (1989), pp. 267–276.

---

[7] It is important to note that we are *not* asserting that every set A that is Turing-reducible to a sparse set is Turing reducible to some sparse set in $\Sigma_2^{p,A}$; the best bound on such sets seems to be $\Delta_3^{p,A}$, via extending [Sch86b, Lemma 5.6] by first taking prefixes and then using adaptive search to find the lexicographically first suitable sparse set (circuit). The somewhat subtle point at work here is that in some cases equivalence allows us to trade off quantifiers between different directions of the equivalence, but reduction allows no such trade-offs.

[IT89] R. Impagliazzo and G. Tardos, *Decision versus search problems in super-polynomial time*, in Proc. 30th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society Press, 1989, pp. 222–227.

[Kad89] J. Kadin, $P^{NP[\log n]}$ *and sparse Turing-complete sets for NP*, J. Comput. System Sci., 39 (1989), pp. 282–298.

[KL80] R. Karp and R. Lipton, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th ACM Symposium on Theory of Computing, April 1980, pp. 302–309.

[Ko] K. Ko, *On adaptive versus nonadaptive bounded query machines*, Theoret. Comput. Sci., 82 (1991), pp. 51–69.

[Ko89] ———, *Distinguishing conjunctive and disjunctive reducibilities by sparse sets*, Inform. Comput., 81 (1989), pp. 62–87.

[LLS75] R. Ladner, N. Lynch, and A. Selman, *A comparison of polynomial time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103–124.

[Mah82] S. Mahaney, *Sparse complete sets for NP: Solution of a conjecture of Berman and Hartmanis*, J. Comput. System Sci., 25 (1982), pp. 130–143.

[Mah86] ———, *Sparse sets and reducibilities*, in Studies in Complexity Theory, R. Book, ed., John Wiley and Sons, New York, 1986, pp. 63–118.

[Mah89] ———, *The isomorphism conjecture and sparse sets*, in Computational Complexity Theory, J. Hartmanis, ed., Proc. Symposia in Applied Mathematics, 38, American Mathematical Society, Providence, RI, 1989, pp. 18–46.

[MP79] A. Meyer and M. Paterson, *With what frequency are apparently intractable problems difficult?*, Tech. Report MIT/LCS/TM-126, MIT Laboratory for Computer Science, Massachusetts Insitute of Technology, Cambridge, MA, 1979.

[PY84] C. Papadimitriou and M. Yannakakis, *The complexity of facets (and some facets of complexity)*, J. Comput. System Sci., 28 (1984), pp. 244–259.

[Sch86a] U. Schöning, *Complete sets and closeness to complexity classes*, Math. Systems Theory, 19 (1986), pp. 29–42.

[Sch86b] ———, *Complexity and Structure*, Lecture Notes in Computer Science, 211, Springer Verlag, Berlin, New York, 1986.

[Sew83] V. Sewelson, *A Study of the Structure of NP*, Ph.D. thesis, Cornell University, Ithaca, NY, 1983. Available as Cornell Department of Computer Science Technical Report #83-575.

[TB] S. Tang and R. Book, *Reducibilities on tally and sparse sets*, RAIRO Inform. Théor. Appl., 25 (1991), pp. 293–302.

[Yes83] Y. Yesha, *On certain polynomial-time truth-table reducibilities of complete sets to sparse sets*, SIAM J. Comput., 12 (1983), pp. 411–425.

SIAM J. COMPUT.
Vol. 21, No. 3, pp. 540–570, June 1992

©1992 Society for Industrial and Applied Mathematics
008

RAY SHOOTING AND OTHER APPLICATIONS OF SPANNING TREES WITH LOW STABBING NUMBER*

PANKAJ K. AGARWAL†

Abstract. This paper considers the following problem: Given a set $\mathcal{G}$ of n (possibly intersecting) line segments in the plane, preprocess it so that, given a query ray ρ emanating from a point p, one can quickly compute the intersection point $\Phi(\mathcal{G}, \rho)$ of ρ with a segment of $\mathcal{G}$ that lies nearest to p. The paper presents an algorithm that preprocesses $\mathcal{G}$, in time $O(n^{3/2} \log^{\omega} n)$, into a data structure of size $O(n\alpha(n) \log^4 n)$, so that for a query ray ρ, $\Phi(\mathcal{G}, \rho)$ can be computed in time $O(\sqrt{n\alpha(n)} \log^2 n)$, where ω is a constant < 4.33 and $\alpha(n)$ is a functional inverse of Ackermann's function. If the given segments are nonintersecting, the storage goes down to $O(n \log^3 n)$ and the query time becomes $O(\sqrt{n} \log^2 n)$. The main tool used is spanning trees (on the set of segment endpoints) with low stabbing number, i.e., with the property that no line intersects more than $O(\sqrt{n})$ edges of the tree. Such trees make it possible to obtain faster algorithms for several other problems, including implicit point location, polygon containment, and implicit hidden surface removal.

Key words. arrangements, fractional cascading, point location, ray shooting, spanning tree, stabbing number, zone

AMS(MOS) subject classifications. 52A37, 68Q20, 68Q25, 68R99

1. Introduction. In the last few years many efficient randomized algorithms, based on the random sampling techniques of [Cl] or on the related ϵ-net theory [HW], have been developed to solve efficiently a variety of geometric problems. One such recent development is due to Welzl [We] (see also [CW]), who showed that, for a given set S of n points in the plane, there exists a spanning tree T of S, such that no line intersects more than $O(\sqrt{n} \log n)$ edges of T. Such a tree T is called a spanning tree with *low stabbing number* (a formal definition is given in §2). Welzl used spanning trees with low stabbing number to obtain an almost optimal algorithm for *simplex range searching*, namely, given a set S of n points in the plane, preprocess it into a data structure of linear size so that, for a query triangle $\triangle$, one can quickly count (or more generally report) all points of S lying inside $\triangle$. His algorithm counts (respectively, reports) the points lying inside a query triangle $\triangle$ in time $O(\sqrt{n} \log^2 n)$ (respectively, $O(\sqrt{n} \log^2 n + K)$, where K is the number of points inside $\triangle$). Soon after this paper, Edelsbrunner et al. [EGH*] used these trees to preprocess a given set $\mathcal{L}$ of n lines in the plane into a data structure of size $O(n \log n)$ so that, for a query point p, the face of the arrangement $\mathcal{A}(\mathcal{L})$ containing p can be computed quickly. The main challenge in both of these papers was to use only roughly linear space (i.e., $O(n \log^{O(1)} n)$ space), because if we allow quadratic space, then a query can be easily answered in $O(\log n)$ time [Ed], [EOS], [EG].

In this paper we present several new applications of spanning trees with low stabbing number. The algorithms presented in this paper are faster than the previously best known algorithms for these problems. One of the main goals of this paper is to demonstrate that such a spanning tree is a versatile tool that can be applied to obtain efficient algorithms for a large class of problems, much beyond the simplex range searching problem for which they were originally introduced. We also show that by combining the spanning tree data structure with the recent partitioning algorithm of [Aga] and [Agb], we can

*Received by the editors June 12, 1989; accepted for publication (in revised form) July 16, 1991. Work on this paper was done while the author was at the Courant Institute of Mathematical Sciences, New York University, New York, New York 10012. This work has been supported by Office of Naval Research grant N00014-87-K-0129, by National Science Foundation grant DCR-83-20085, and by grants from the Digital Equipment Corporation and the IBM Corporation.

† Computer Science Department, Duke University, Durham, North Carolina 27706.

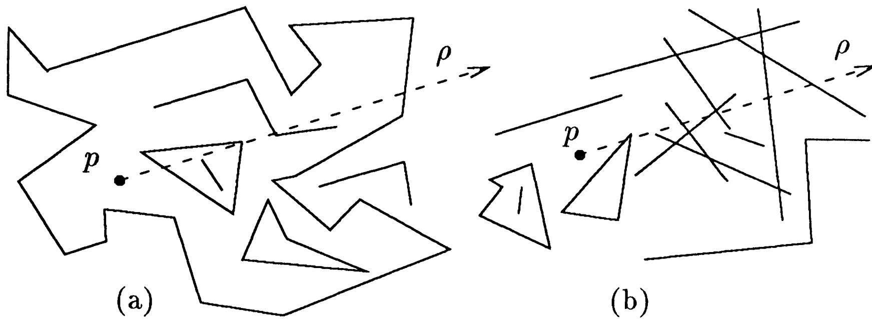


FIG. 1. *Ray shooting in an arrangement of* (a) *nonintersecting and* (b) *arbitrary segments.*

obtain a trade-off between space and query time. Similar trade-offs have been obtained earlier [EGH*], [Agc], [Chd].

The first and perhaps the most interesting application that we consider is *ray shooting* in arrangements of segments. There are two versions of this problem, one for segments that are nonintersecting, and one for an arbitrary collection of segments. Formally, these problems can be stated as follows:

(a) *Given a collection $\mathcal{G} = \{e_1, \cdots, e_n\}$ of n nonintersecting line segments in the plane, preprocess it so that, given a query ray ρ emanating from a point p in direction d, we can quickly compute the intersection point $\Phi(\mathcal{G}, \rho)$ of ρ with the segments of $\mathcal{G}$ that lies nearest to p (see Fig.* 1(a)).

(b) *Same problem, except that the segments in $\mathcal{G}$ can intersect arbitrarily* (*see Fig.* 1(b)).

If the segments in $\mathcal{G}$ form the boundary of a simply connected region, then the algorithm of Chazelle and Guibas [CGa] preprocesses $\mathcal{G}$ into a data structure of linear size so that, for any ray ρ, $\Phi(\mathcal{G}, \rho)$ can be computed in $O(\log n)$ time (see also [GHLST]). For the general case, however, the ray shooting and other visibility problems are much harder even for nonintersecting segments. For example, a result of Suri and O'Rourke [SO] shows that the portion of a polygon, with holes, visible from a fixed edge can have $\Omega(n^4)$ edges on its boundary, while for simple polygons such a region is bounded by only $O(n)$ edges.

We are not aware of any ray shooting algorithm for nonsimple polygons (or for an arrangement of segments), which answers a query in $O(\log^{O(1)} n)$ time, using roughly linear space. If we allow quadratic space, then a query is easy to answer in time $O(\log n)$ (see §4.1). Our goal in this paper is to obtain efficient solutions that use roughly linear space, and to establish a trade-off between space and query time.

For a special case, where $\mathcal{G}$ is a set of lines, a result of Edelsbrunner et al. [EGH*] implies that we can construct, in randomized expected time $O(n^{3/2} \log^2 n)$, a data structure of size $O(n \log^2 n)$, so that a ray shooting query in $\mathcal{A}(\mathcal{L})$ can be answered in $O(\sqrt{n} \log^3 n)$ time. (The preprocessing has been made deterministic and the query time has been reduced to $O(\sqrt{n} \log n)$ in [Agc].) Unfortunately, this algorithm does not apply to segments. An algorithm with a sublinear query time for the case of segments can be developed using the "recursive space-cutting tree" of Dobkin and Edelsbrunner [DE] (see also [EW]). The best-known algorithm for computing $\Phi(\mathcal{G}, \rho)$ is by Guibas et al. [GOS], which constructs a data structure of size $O(n)$, so that a query can be answered in

$O(n^{2/3+\delta})$ time, for any $\delta > 0$. Their algorithm is based on the random sampling technique of [Cl] and [HW], and constructs a multilevel partition tree. The preprocessing of their algorithm is randomized with $O(n \log n)$ expected running time. However, the preprocessing can be made deterministic without any additional overhead using the recent partitioning algorithms of Matoušek [Maa] or Agarwal [Agb].

In this paper we show that ray shooting can be performed in roughly (that is, up to polylogarithmic factors) $\sqrt{n}$ time, while still using only roughly linear space and employing deterministic, rather than randomized, preprocessing techniques. We first give an algorithm for the case of nonintersecting segments. This algorithm constructs, in time $O(n^{3/2} \log^{\omega} n)$, a data structure of size $O(n \log^3 n)$ so that, for a given ray ρ, $\Phi(\mathcal{G}, \rho)$ can be computed in $O(\sqrt{n} \log^2 n)$ time, where ω is a constant less than 4.33. Our algorithm is simpler than that of [GOS] because it maintains only a two-level data structure. We then extend the above algorithm to general arrangements of segments. Although the basic idea remains the same, we need several new techniques, and the algorithm is more complex. In this case a query can be answered in $O(\sqrt{n\alpha(n)} \log^2 n)$ time, using $O(n\alpha(n) \log^4 n)$ space, after $O(n^{3/2} \log^{\omega} n)$ preprocessing. Another major difference between the two cases is that in the first case we can report all K intersections between a query ray ρ and $\mathcal{G}$ in $O(\sqrt{n} \log^2 n + K \log n)$ time, while we still do not know how to report these intersections in a comparably efficient manner in the general case. One disadvantage of our algorithms over those of [GOS] and [DE] is that our preprocessing time is roughly $n^{3/2}$ instead of roughly linear. This is the price that we must pay to achieve deterministic preprocessing and to reduce the query time.

The second problem for which we give an efficient algorithm using the spanning tree data structure is *implicit point location*. The implicit point location problem is an extension of the widely studied planar point location problem (see [Ki], [EGS], and [ST]). In the latter problem, a planar map M consisting of n faces is given, and the goal is to preprocess M into a data structure that supports fast point location queries, i.e., queries that seek the face of M containing a query point p. The above algorithms construct, in time $O(n \log n)$ (or sometimes linear), a data structure of linear size, so that a query point can be located in M in $O(\log n)$ time. In the implicit point location problem the map is defined implicitly. In particular, we assume that it is defined as the arrangement (i.e., overlay) of a given set of n geometric polygonal (possibly intersecting) objects of some simple shape (or as a collection of arbitrary line segments), and the goal is to obtain certain information related to the arrangement of the objects; for example, to determine whether a query point lies in the union of the objects. A more formal description is given in §7. Guibas et al. [GOS] have presented an algorithm with $O(n^{2/3+\delta})$ query time, for any $\delta > 0$, using the random sampling technique. We improve the query time to $O(\sqrt{n} \log^2 n)$ and use deterministic preprocessing. The algorithm of [GOS] uses $O(n)$ space, while ours requires $O(n \log^2 n)$ space.

Guibas et al. [GOS] have described several applications of the implicit point location problem, such as polygon containment, implicit hidden surface removal, polygon placement, etc. We show that our implicit point location algorithm improves the query time of these algorithms too.

This paper is organized as follows. In §2 we discuss spanning trees with low stabbing number. Section 3 describes our ray shooting algorithm for arrangements of nonintersecting segments. In §4 we show that ray shooting queries can be performed faster, if we are allowed to use more space. Section 5 extends the algorithms of §§3 and 4 to report all intersections between $\mathcal{G}$ and a query ray ρ at logarithmic cost per intersection. In §6 we generalize our ray shooting algorithms to arrangements of arbitrary (possibly inter-

secting) segments. Section 7 presents an efficient algorithm for implicit point location and §8 discusses other applications of the spanning tree data structure. We conclude in §9 with some final remarks.

2. Spanning trees of low stabbing number. Let S be a set of n points in $\mathbb{R}^d$, and let $\mathcal{T}$ be a spanning tree of S whose edges are line segments. The *stabbing number* $\sigma(\mathcal{T})$ of $\mathcal{T}$ is the maximum number of edges of $\mathcal{T}$ that can be intersected by a hyperplane h. Chazelle and Welzl [CW] (see also [Chc], [We]) have proved that, for any set of n points in $\mathbb{R}^d$, there exists a spanning tree with stabbing number $O(n^{1-1/d})$, and that this bound is tight in the worst case. For a family $\mathbf{T}$ of trees, the stabbing number $\sigma(\mathbf{T})$ is s if for each hyperplane h there is a tree $\mathcal{T} \in \mathbf{T}$ such that h intersects at most s edges of $\mathcal{T}$.

Chazelle and Welzl [CW] also proved that a spanning tree of n points in $\mathbb{R}^d$ with stabbing number $O(n^{1-1/d})$ can be constructed in polynomial time. In the plane, a spanning tree with stabbing number $O(\sqrt{n})$ can be constructed in $O(n^3 \log n)$ time. A recent algorithm of Matoušek [Mab] improves the running time to $O(n^{5/2} \log^2 n)$ at the cost of increasing $\sigma(\mathcal{T})$ to $O(\sqrt{n} \log n)$. As for constructing a family of spanning trees, Edelsbrunner et al. [EGH*] have presented a randomized algorithm, with expected running time $O(n^{3/2} \log^2 n)$, to compute a family $\mathbf{T} = \{\mathcal{T}_1, \cdots, \mathcal{T}_k\}$ of $O(\log n)$ spanning trees, with $\sigma(\mathbf{T}) = O(\sqrt{n} \log^2 n)$. The running time of their algorithm has been improved to $O(n^{4/3} \log^2 n)$ in another randomized algorithm by Matoušek [Mab]. (The stabbing number of $\mathbf{T}$ computed by Matoušek's algorithm can actually be improved to $O(\sqrt{n} \log n)$; see [Agc].) An additional property of the algorithms of [EGH*] and [Mab] is that the trees they produce are actually spanning paths. The best known deterministic algorithm for constructing a family of spanning path is due to Agarwal [Agc], who has shown the following.

THEOREM 2.1. [Agc] *Given a set S of n points in the plane, we can deterministically construct a family $\mathbf{C}$ of $O(\log n)$ spanning paths on S with $\sigma(\mathbf{C}) = O(\sqrt{n})$, in $O(n^{3/2} \log^{\omega} n)$ time, using $O(n^{3/2})$ working storage, where ω is a constant less than 4.33. Moreover, for any query line ℓ, we can determine in $O(\log n)$ time a spanning path $\mathcal{C} \in \mathbf{C}$ such that ℓ intersects at most $O(\sqrt{n})$ edges of $\mathcal{C}$.*

The paths constructed by [Mab] and [Agc] can generally be self-intersecting. However, Edelsbrunner et al. [EGH*] have shown that a spanning tree $\mathcal{T}$ can be converted into a simple polygonal path $\mathcal{C}$ in $O(n \log n)$ time, so that if a line ℓ intersects s edges of $\mathcal{T}$, then ℓ intersects at most $2s$ edges of $\mathcal{C}$. Therefore, if desired, we can assume that the spanning paths produced by the techniques of [Mab] and [Agc] are non–self-intersecting.

Let $\mathcal{C}$ be a spanning path on S. For our applications we need to construct a balanced binary tree $\mathcal{B}$ on $\mathcal{C}$ whose leaves store the points of S in their order along $\mathcal{C}$. Each node v of $\mathcal{B}$ is associated with the subpath $\mathcal{C}_v$ of $\mathcal{C}$ connecting the points stored at the leaves of the subtree rooted at v; let us denote by S_v the subset of S consisting of these points (see Fig. 2).

A line ℓ *stabs* a node v of $\mathcal{B}$ if ℓ intersects $\mathcal{C}_v$. Let $V_{\mathcal{B}}(\ell)$ denote the set of nodes v of $\mathcal{B}$ such that v is not stabbed by ℓ but its parent (if one exists) is stabbed. It is easily seen that $\{S_v : v \in V_{\mathcal{B}}(\ell)\}$ is a disjoint partitioning of S. Moreover, we have the following lemma.

LEMMA 2.2. *If a line ℓ intersects s edges of $\mathcal{C}$, then $|V_{\mathcal{B}}(\ell)| \leq 2(s+1) \log n$ and the nodes of $V_{\mathcal{B}}(\ell)$ lie on at most $2(s+1)$ paths of $\mathcal{B}$.*

Another simple but key observation is given in the following lemma.

LEMMA 2.3. *A line ℓ intersects a polygonal path $\mathcal{C}$ if and only if ℓ intersects the convex hull of the vertices of $\mathcal{C}$.*

Lemma 2.3 implies that ℓ stabs a node v if and only if ℓ intersects the convex hull

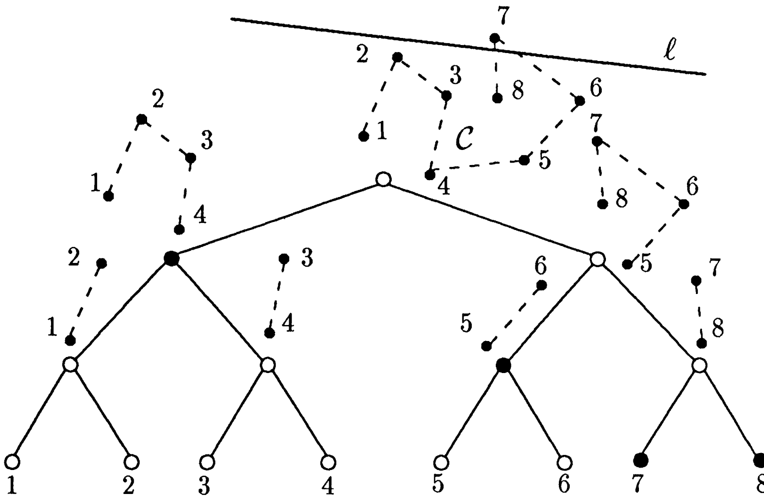


FIG. 2. *$\mathcal{C}$ and $\mathcal{B}(\mathcal{C})$: black nodes of $\mathcal{B}$ denote $V_{\mathcal{B}}(\ell)$.*

of S_v. Since an intersection between a line and a convex n-gon can be detected in $O(\log n)$ time, it follows that $V_{\mathcal{B}}(\ell)$ and the paths containing its nodes can be computed in $O(|V_{\mathcal{B}}(\ell)| \log n)$ time, if we store the convex hull of the subpath $\mathcal{C}_v$ at each node v of $\mathcal{B}$. The running time of this computation can actually be improved to time $O(|V_{\mathcal{B}}(\ell)| + \log n)$, using fractional cascading (cf. [CGc]).

All the problems considered in this paper involve a set of segments in $\mathbb{R}^2$ and most of the algorithms presented here are based on spanning paths with low stabbing number. The spanning path is constructed either on the endpoints of the segments or on the points dual to the lines containing the segments. To answer a query, we choose a line ℓ depending on the query and the problem (e.g., in the ray shooting problem, we take ℓ to be the line containing the query ray), and compute the intersection points of ℓ and the spanning path. The portion π of the spanning path between two consecutive intersection points lies either above or below ℓ. The query for segments corresponding to the points lying on π is answered directly in $O(\log^{O(1)} n)$ time, see below for details. We repeat this procedure for all such portions of the spanning path and then compute the overall answer from them. If ℓ intersects s edges of the path, the query time is $O(s \log^{O(1)} n)$. Since $s = O(\sqrt{n})$, the query time of these algorithms will be $O(\sqrt{n} \log^{O(1)} n)$.

3. Ray shooting in arrangements of nonintersecting segments. In this section we present an algorithm that preprocesses a given set $\mathcal{G}$ of n nonintersecting segments so that, given a query ray ρ emanating from a point p in direction d, $\Phi(\mathcal{G}, \rho)$ can be computed quickly. (For technical reasons we consider ρ as an open ray, i.e., the point p does not belong to ρ.) We will also use $\Phi(\mathcal{G}, \rho)$ to denote the distance of that point from p; if no such intersection exists, we put $\Phi(\mathcal{G}, \rho) = +\infty$. Without loss of generality, we restrict our attention to rightward-directed rays; leftward-directed rays can be handled in a symmetric way. We also assume that there is no vertical segment in $\mathcal{G}$. Denote the set of left endpoints of the segments of $\mathcal{G}$ as $S = \{p_1, \cdots, p_m\}$, where $m \leq n$. Let $\mathbf{C} = \{\mathcal{C}_1, \cdots, \mathcal{C}_k\}$ denote a family of $k = O(\log n)$ spanning paths on S, with $\sigma(\mathbf{C}) = O(\sqrt{n})$.

We show how to preprocess a single path $\mathcal{C} \in \mathbf{C}$. First, construct the binary tree $\mathcal{B} = \mathcal{B}(\mathcal{C})$. Let $\mathcal{G}_v \subseteq \mathcal{G}$ be the set of segments whose left endpoints are in S_v (see Fig. 3).

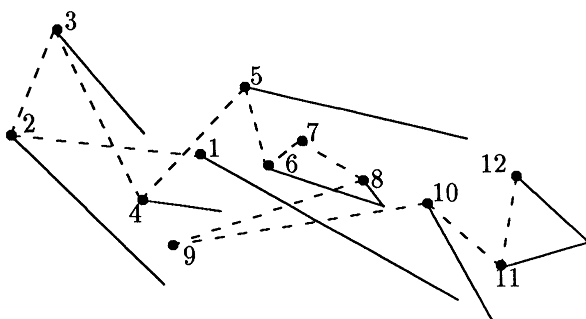


FIG. 3. $\mathcal{G}_v$: *dashed path denotes* C_v; *solid lines denote* $\mathcal{G}_v$; *bullets denote* S_v.

Let ℓ denote the line containing the query ray ρ; then

$$\Phi(\mathcal{G}, \rho) = \min_{v \in V_{\mathcal{B}}(\ell)} \{ \Phi(\mathcal{G}_v, \rho) \}. \tag{3.1}$$

Note that for a node $v \in V_{\mathcal{B}}(\ell)$ C_v is a connected path; therefore, either all points in S_v lie above ℓ or all of them lie below ℓ. In what follows we assume that all points of S_v lie above ℓ. We will show below that $\Phi(\mathcal{G}_v, \rho)$ can be computed in $O(\log n)$ time. First, a few notations: Let ℓ^- (respectively, ℓ^+) denote the half plane lying below (respectively, above) the line ℓ. We distinguish between the two sides of a segment e, the top (respectively, bottom) side of e is denoted by e^+ (respectively, e^-). We say that a ray ρ hits e from *above* (respectively, *below*) if slightly to the left of their intersection, ρ lies above (respectively, below) e. If we think of e as expanded into a very thin rectangle and of e^+, e^- as denoting the top and bottom sides of that rectangle, respectively, then ρ hits e from above if, when traversed from left to right, ρ first intersects e^+ and then e^-, and symmetrically for rays that hit e from below (see Fig. 4). If ρ hits e from above (respectively, below), then we also say that it hits e^+ (respectively, e^-). The following lemma is quite obvious, so we state it without proof.

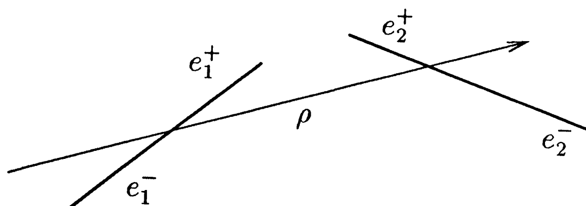


FIG. 4. *Two-sided segments:* ρ *hits* e_1 *from above and* e_2 *from below.*

LEMMA 3.1. *Let v be a node of $V_{\mathcal{B}}(\ell)$. Under the assumption that all points of S_v lie above ℓ, if ρ hits a segment $e \in \mathcal{G}_v$, then it hits e from below.*

Before proceeding, we introduce a linear ordering among the segments of $\mathcal{G}_v$, as defined in [GOS] (see also [GY]). As we will see later, this ordering sorts the segments in a manner that is consistent with any order in which they can be crossed by a rightward-directed ray (from below).

DEFINITION 3.2 [GOS]. For a given set $\mathcal{G} = \{e_1, \cdots, e_n\}$ of segments,

(i) $e_i <_s e_j$ if there exists a (nonvertical) line ℓ hitting both e_i^- and e_j^- such that its intersection with e_i lies to the left of its intersection with e_j, and such that ℓ does not hit any e_k^+, for $k \neq i, j$, at a point between e_i and e_j.

(ii) $e_i <_v e_j$ if there exists a vertical line intersecting both e_i and e_j such that its intersection with e_i lies below its intersection with e_j.

(iii) $e_i <_x e_j$ if e_i and e_j have nonoverlapping x-projections and the projection of e_i lies to the left of that of e_j.

(iv) $e_i < e_j$ if either e_i precedes e_j in the transitive closure $<_{v^*}$ of $<_v$, or e_i and e_j are not related by $<_{v^*}$ and $e_i <_x e_j$.

THEOREM 3.3 [GOS]. *$<_s$ is a partial order, and $<$ is a linear order that extends $<_s$. Moreover $<$ can be computed in time $O(n \log n)$.*

Remark 3.4. It is possible for a pair of segments e_1, e_2 that $e_2 < e_1$ within some set $\mathcal{G}$, but $e_1 < e_2$ relative to a subset $\mathcal{G}' \subset \mathcal{G}$ (see Fig. 5). Therefore, it is important to mention the set relative to which we are ordering the segments.

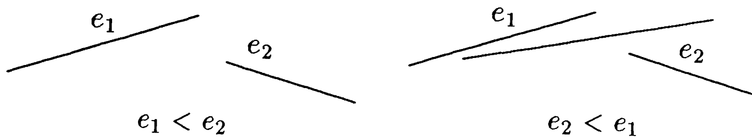


FIG. 5. *Ordering of a pair of segments is relative to a set.*

Next we prove a technical lemma about $<$ that we will need later. Let l_e (respectively, r_e) denote the left (respectively, right) endpoint of a nonvertical segment e.

LEMMA 3.5. *For all segments $a, b \in \mathcal{G}_v$, if r_a lies below ℓ and $x(r_a) < x(l_b)$, then $a < b$ (relative to $\mathcal{G}_v$).*

Proof. Suppose, to the contrary that, there is a pair of segments $a, b \in \mathcal{G}_v$ such that r_a lies below ℓ and $x(r_a) < x(l_b)$, but $b < a$. Since the x-projection of b is to the right of that of a, the only way b can precede a in $<$-ordering is by the transitive relation $<_{v^*}$. Thus there exists a sequence of segments in $\mathcal{G}_v$ such that $b = e_1 <_v e_2 <_v \cdots <_v e_k = a$. Let $\pi_{b,a}$ denote a shortest sequence among all such sequences, and let $d_{b,a}$ denote the length of $\pi_{b,a}$. We obtain a contradiction by showing that, for every $k > 0$, there is no sequence $\pi_{b,a}$ such that $d_{b,a} = k$.

Obviously $d_{b,a} > 2$, because $x(r_a) < x(l_b)$. If $d_{b,a} = 3$, then there is a segment $c \in \mathcal{G}_v$ such that $b <_v c <_v a$. This implies that $x(r_c) \geq x(l_b) > x(r_a) \geq x(l_c)$. Let q be the intersection point of c and the vertical line $x = x(r_a)$. Note that a and c satisfy the following properties: (i) $c <_v a$, (ii) $x(r_c) > x(r_a)$, (iii) a does not intersect c, and (iv) the point l_c lies above ℓ (because $c \in \mathcal{G}_v$). These properties imply that $\overline{qr_c}$ lies below ℓ (see Fig. 6), which contradicts the assumption that $b <_v c$ (because $x(r_c) \geq x(l_b)$ and c lies below b at $x = x(l_b)$). Hence $d_{b,a} > 3$.

Now assume that, for all segments $a, b \in \mathcal{G}_v$ satisfying the conditions of the lemma, either $a < b$ or $d_{b,a} \geq k$. Suppose there exists a pair a, b such that $b <_{v^*} a$ and $d_{b,a} = k$. Let $b = e_1 <_v \cdots <_v e_{k-1} <_v e_k = a$ be a corresponding shortest sequence $\pi_{b,a}$, and let $c = e_{k-1}$. Since $\pi_{b,a}$ is a shortest sequence, it is easily seen that $x(r_c) > x(r_a)$. Indeed, let e_j be the first segment in this sequence whose x-projection overlaps that of a. Then e_j must lie below a, for otherwise we would have obtained a cycle in $<_{v^*}$,

which is impossible. Hence $e_j <_v a$ and we can shortcut the sequence after e_j. Clearly, $x(r_j) > x(r_a)$. Let q be the intersection of c with $x = x(r_a)$, as above. Again we can argue that $\overline{qr_c}$ lies below ℓ. If $x(r_c) < x(l_b)$, then c and b satisfy the property of the lemma, and thus contradict the inductive hypothesis because $d_{b,c} < k$. On the other hand, if $x(r_c) \geq x(l_b)$, then we have $c <_v b$ (because c lies below b at $x = x(l_b)$), contradicting the assumption that $b <_{v^*} c$. Hence, we can conclude that $a < b$, and this completes the proof. □

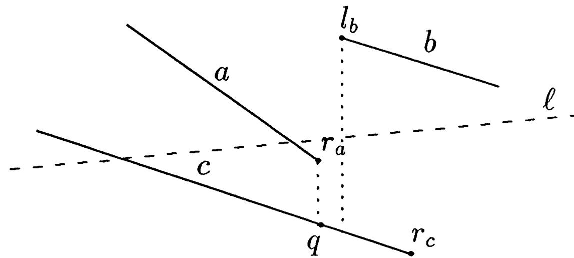


FIG. 6. *Illustration for Lemma* 3.5, $d_{b,a} = 3$.

Using Lemma 3.1 and Theorem 3.3 we obtain the following lemma.

LEMMA 3.6. *Let* $\mathcal{E} = (e_1, \cdots, e_m)$ *denote the segments of* $\mathcal{G}_v$ *ordered with respect to* $<$ *(relative to* $\mathcal{G}_v$*), and suppose* $\Phi(\mathcal{G}_v, \rho) = \rho \cap e_f$, *for some* $1 \leq f \leq m$. *Then for all* $i < f$, e_i *does not intersect* ρ.

Proof. If e_i intersects ρ, it does so at a point to the right of $\rho \cap e_f$. This implies that $e_f <_s e_i$, which means $e_f < e_i$ since $<$ extends $<_s$. □

Hence the original problem is reduced to the following restricted problem:

Given a sequence $\mathcal{E}$ *of* m *segments sorted according to* $<$, *preprocess* $\mathcal{E}$ *so that for any (rightward-directed) query ray* ρ *emanating from a point* p *and lying on a line that passes below the left endpoints of all segments in* $\mathcal{E}$, *we can quickly determine* $e_f(\rho)$, *the first segment of* $\mathcal{E}$ *hit by the ray* ρ.

A possible approach to solving this problem is to do a binary search on $\mathcal{E}$, where each step of the search tests whether ρ intersects a segment in some contiguous subsequence of t segments of $\mathcal{E}$. If ρ were a full line ℓ, then such an intersection could be easily detected in $O(\log t)$ time after $O(t \log t)$ preprocessing (in which we construct the convex hull of the right endpoints of these t segments). However, no equally fast procedure is known to detect an intersection between a ray and such a set of segments. To overcome this problem, we next show how to reduce the intersection detection problem to one involving the line containing ρ rather than ρ itself.

For any point q in the plane, let $e_h = e_{h(q)}$ denote the first segment of $\mathcal{E}$ whose left endpoint lies to the right of (or above) q (see Fig. 7), and let $e_u = e_{u(q)}$ denote the segment in $\mathcal{E}$ lying immediately above q, that is, the vertical ray emanating from q in the upward direction hits e_u before any other segment. If e_h (respectively, e_u) is not defined, we put $h = m + 1$ (respectively, $u = m + 1$). Finally, put $\phi_q = \min\{h, u\}$.

To compute e_h, construct a balanced binary tree L whose leaves store the segments of $\mathcal{E}$ in their order in $\mathcal{E}$. For each interior node z of L we store the rightmost left endpoint of the segments stored at the subtree rooted at z. L can be constructed in $O(m \log m)$ time, and e_h can be determined, by searching for q through L, in $O(\log m)$ time. As for e_u, we can easily calculate it in time $O(\log m)$ after $O(m \log m)$ preprocessing, as in [ST].

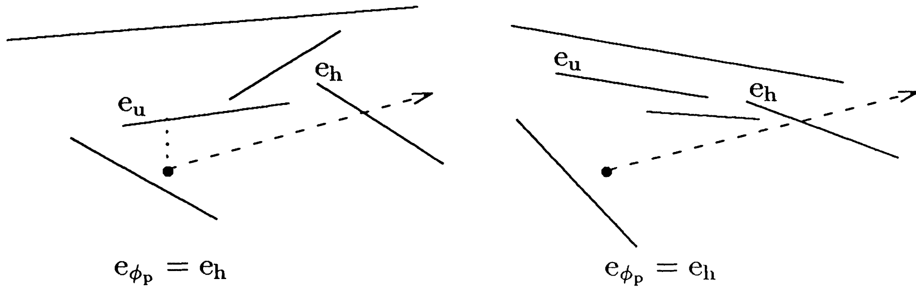


FIG. 7. *Segments e_h, e_u, and e_{ϕ_p}.*

LEMMA 3.7. *The query ray ρ emanating from a point p cannot intersect any segment $e_i \in \mathcal{E}$ for $i < \phi_p$. Moreover, ρ intersects e_i for $i \geq \phi_p$ if and only if its right endpoint lies below the line containing ρ.*

Proof. If the first part of the lemma were not true, then there would exist a segment e_i for $i < \phi_p$, intersecting the ray ρ. In this case the left endpoint of e_i must lie to the left of p, so the vertical ray η from p in the upward direction must intersect e_i. But then the first segment e_u hit by η must satisfy $i \geq u \geq \phi_p$ (because $e_u <_{v^*} e_i$ and by definition of ϕ_p), a contradiction that proves the first half of Lemma 3.7.

The "only if" part of the second half of Lemma 3.7 follows from the fact that if both the left and the right endpoints of a segment e lie above ℓ, then e cannot intersect ℓ. For the "if" part let e_i be a segment of $\mathcal{E}$, for $i \geq \phi_p$, whose right endpoint lies below ℓ, but e_i does not intersect ρ. If the left endpoint l_{e_i} of e_i lies to the right of p, then obviously e_i intersects ρ, so l_{e_i} must lie to the left of p. Since e_i does not intersect ρ, the intersection point ξ of e_i and ℓ lies to the left of p. Moreover if $x(r_{e_i}) < x(l_{e_{u(p)}})$, then by Lemma 3.5 $e_i < e_{u(p)}$. If $x(r_{e_i}) \geq x(l_{e_{u(p)}})$, then e_i and $e_{u(p)}$ must have x-projections that overlap at some point between ξ and p; since e_i lies below $e_{u(p)}$ at this point, we again have $e_i <_{v^*} e_{u(p)}$. Similarly we can show that $e_i < e_{h(p)}$. Hence $i < \min\{u, h\}$, contradicting the assumption that $i \geq \phi_p$. □

Lemma 3.7 implies that the binary search technique proposed above will work, provided we can detect quickly whether the right endpoint of any segment in some contiguous subsequence of $\mathcal{E}$ lies below ℓ. In other words, the problem now has been reduced to that of detecting an intersection between a set of points and a query half plane. Clearly, this is equivalent to detecting an intersection between the convex hull of these points and the half plane (see Fig. 8).

We are now ready to describe how to preprocess $\mathcal{E}$ so that $e_f(\rho)$, the first segment of $\mathcal{E}$ hit by ρ, can be computed quickly, for any ray ρ with the above properties. Let r_i denote the right endpoint of $e_i \in \mathcal{E}$, and let $R = \{r_1, \cdots, r_m\}$. We construct a binary tree $\mathcal{T}$ on R in the same way as we constructed $\mathcal{B}$, i.e., the points r_i are stored at the leaves of $\mathcal{T}$ in order, and each node w of $\mathcal{T}$ is associated with the subsequence R_w of R containing all points stored at the leaves of the subtree rooted at w.

At every node w of $\mathcal{T}$, we store the convex hull of R_w. Using $\mathcal{T}$ we can determine $e_f(\rho)$ in time $O(\log^2 m)$ as follows: We first find ϕ_p, as described above, in $O(\log m)$ time. Then we treat the suffix $\{r_{\phi_p}, \cdots, r_m\}$ of R as the union of $\log m$ subsets R_w, $w \in \mathcal{T}$, which we can compute in $O(\log m)$ time. We test each R_w in increasing, left-to-right order, to find the first w for which the line ℓ containing ρ intersects the hull of R_w. Then we do a binary search within R_w until we find $e_f(\rho)$. All this takes $O(\log^2 m)$ time.

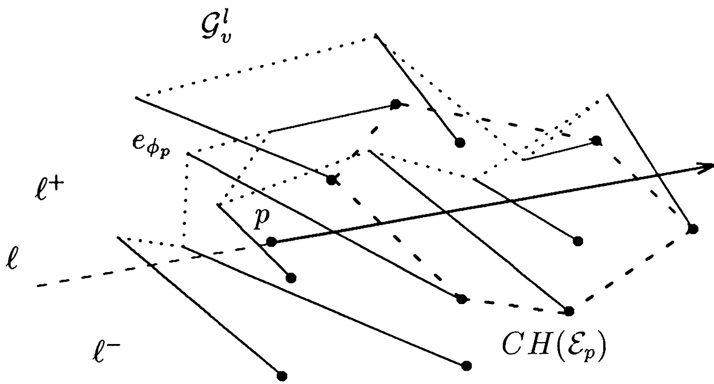


FIG. 8. *Convex hull* $\mathrm{CH}(\mathcal{E}_p)$ *intersecting* ℓ^-.

However, we can easily reduce the time to $O(\log m)$, using fractional cascading. This is possible since, as in [CGc], detecting intersection between ℓ and a convex polygon amounts to searching for the slope of ℓ in the sequence of slopes of the edges of the polygon (see [CGb] and [CGc] for more details). We thus have the following lemma.

LEMMA 3.8. *Given a set $\mathcal{E}$ of m nonintersecting line segments in the plane, we can preprocess it, in time $O(m \log m)$, into a data structure of size $O(m \log m)$ so that, given a (rightward-directed) query ray ρ whose containing line lies below the left endpoints of all the segments in $\mathcal{E}$, we can compute the first segment of $\mathcal{E}$ hit by ρ in time $O(\log m)$.*

Returning to the original problem, Lemma 3.8 and the preceding discussion imply that we can compute $\Phi(\mathcal{G}_v, \rho)$ for each $v \in V_{\mathcal{B}}(\ell)$, in time $O(\log n)$. Equation (3.1) and Lemma 2.2 then imply the following theorem.

THEOREM 3.9. *Given a set $\mathcal{G}$ of n nonintersecting line segments, we can preprocess it in time $O(n^{3/2} \log^{\omega} n)$ for some $\omega < 4.33$ into a data structure of size $O(n \log^3 n)$, using $O(n^{3/2})$ working storage, so that, given a query ray ρ, its first intersection $\Phi(\mathcal{G}, \rho)$ with $\mathcal{G}$ can be computed in time $O(\sqrt{n} \log^2 n)$.*

Remark 3.10.

(i) The space used can be reduced to $O(n \log^2 n)$ without affecting the query time if we use a single tree structure instead of a family of $O(\log n)$ trees. But then the preprocessing time increases to $O(n^3 \log n)$ (see [EGH*]).

(ii) If we allow randomization, the (expected) preprocessing time of the algorithm can be reduced to $O(n^{4/3} \log^2 n)$ using Matoušek's algorithm for computing a family of spanning trees [Mab], but then the query time bound increases by a factor of $\log n$.

4. Trade-off between space and query time. In this section we show that the query time for the ray shooting problem in arrangements of nonintersecting segments can be improved if we allow ourselves more storage. Similar trade-offs have been obtained for several related problems, such as computing a face in an arrangement of lines [EGH*] and simplex range searching [Agc], [Chd]. The main result of this section is an algorithm for computing $\Phi(\mathcal{G}, \rho)$ with $O(\frac{n}{\sqrt{m}} \log^{7/2} \frac{n}{\sqrt{m}} + \log n)$ query time, using $O(m)$ space, where $n \log^3 n \leq m \leq n^2$.

4.1. The case of quadratic storage. First, we show that if we allow $O(n^2)$ space, the query time can be reduced to $O(\log n)$.

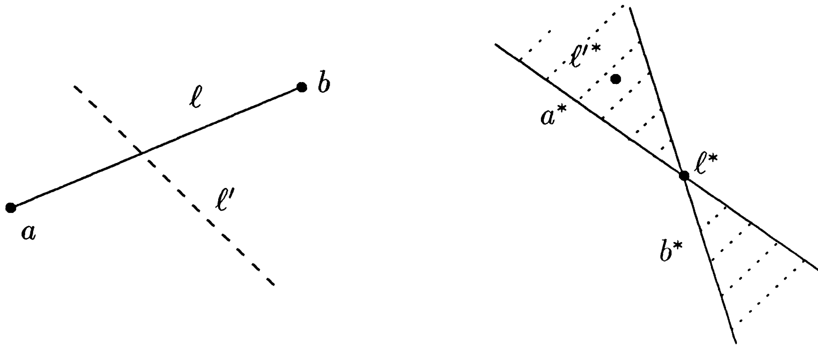


FIG. 9. *A segment e and its dual e^*.*

Let $\mathcal{G} = \{e_1, \cdots, e_n\}$ be a collection of n nonintersecting segments. The dual of a segment $e = \overline{ab}$ is a double wedge e^* formed between the dual lines a^*, b^* of a, b, respectively, and not containing any vertical line (see Fig. 9). Dualize all segments $e \in \mathcal{G}$, obtaining a set $\mathcal{G}^*$ of n double wedges. Let $\mathcal{L}^*$ denote the set of lines bounding the double wedges of $\mathcal{G}^*$ (i.e., the duals of the endpoints of segments in $\mathcal{G}$). Let $\mathcal{A}(\mathcal{L}^*)$ denote the arrangement of $\mathcal{L}^*$, and let w_f be the set of segments dual to the double wedges of $\mathcal{G}^*$ containing the face $f \in \mathcal{A}(\mathcal{L}^*)$. Standard duality arguments yield the following lemma (see, e.g., [CGL]).

LEMMA 4.1. *Let p be a point lying in the interior of a face f of $\mathcal{A}(\mathcal{L}^*)$. Then p^* intersects each segment $e \in w_f$ transversally at an interior point, and is disjoint from any other segment of $\mathcal{G}$.*

LEMMA 4.2. *If the segments of $\mathcal{G}$ are nonintersecting, then for all points p in a face f of $\mathcal{A}(\mathcal{L}^*)$, the line p^* intersects the segments of w_f in the same order.*

Proof. Suppose there are two points x and y in a face f such that the lines x^* and y^* intersect the segments of w_f in two different orders. Since the segments in $\mathcal{G}$ are nonintersecting, rotating x^* towards y^* (in the direction that avoids a vertical orientation) we must reach a line p^* that either contains a segment of w_f, or passes through an endpoint of a segment of w_f. (Note that this claim does not hold if the segments can intersect.) The dual p of p^* is a point that lies on the segment $\overline{xy}$, hence in f. This, however, contradicts Lemma 4.1, thus showing that the duals of all points in f intersect the segments of w_f in the same order. □

Sort the segments in w_f in the order provided by Lemma 4.2. For a ray ρ, let the *image* of ρ be the dual of the line containing ρ. If the image of ρ lies in the face $f \in \mathcal{A}(\mathcal{L}^*)$, then $\Phi(\mathcal{G}, \rho)$ can be computed in $O(\log n)$ time by a binary search on w_f. Therefore, it suffices to show how to store all the lists w_f using only $O(n^2)$ space, so that binary search in each of them can still be done in $O(\log n)$ time.

Let $\mathcal{D}$ denote the dual graph of $\mathcal{A}(\mathcal{L}^*)$, i.e., the graph whose nodes represent faces of $\mathcal{A}(\mathcal{L}^*)$ and whose edges connect pairs of nodes representing adjacent faces. Let $\mathcal{T}$ denote a spanning tree of $\mathcal{D}$. We can convert $\mathcal{T}$ into a path Π by tracing an Eulerian tour around the tree. Observe that if two vertices v_1, v_2 in Π represent faces f_1, f_2 sharing an edge γ, which is a portion of a line ℓ, then $w_{f_1} \oplus w_{f_2}$ is the set of segments having the dual of ℓ as an endpoint. Let δ_γ denote this set of segments. The set w_{f_2} can be obtained from w_{f_1} by deleting the segments of $\delta_\gamma \cap w_{f_1}$ and inserting the segments of $\delta_\gamma - w_{f_1}$.

Therefore, we can maintain all lists w_f using a persistent data structure (see [Co], [ST] and [DSST]). Since at each edge γ of Π only the segments of δ_γ are inserted or deleted, the total space required to store all w_f is $O(n + \sum_{\gamma \in \Pi} |\delta_\gamma|)$ and the total preprocessing time is $O((n + \sum_{\gamma \in \Pi} |\delta_\gamma|) \log n)$. Moreover, using the persistent data structure, $\Phi(\mathcal{G}, \rho)$ can be computed in $O(\log n)$ time (see [ST]). Thus, it suffices to prove that $\sum_\gamma |\delta_\gamma| = O(n^2)$. Suppose the segments of $\mathcal{G}$ have $t \leq 2n$ distinct endpoints and ν_i segments are incident to the ith endpoint. It is easy to check that if γ is a portion of the line dual to the ith endpoint, then $|\delta_\gamma| \leq \nu_i$. Obviously, $\sum_{i=1}^{t} \nu_i = 2n$ and each line of $\mathcal{L}^*$ is split into $\leq t + 1$ edges, which implies that

$$\sum_{\gamma \in \mathcal{D}} |\delta_\gamma| \quad \leq \quad \sum_{i=1}^{t} (t+1)\nu_i \quad \leq \quad 2n(2n+1).$$

Hence, we have the following theorem.

THEOREM 4.3. *Given a collection $\mathcal{G}$ of n nonintersecting segments, we can preprocess it, in $O(n^2 \log n)$ time, into a data structure of size $O(n^2)$ so that, for any query ray ρ, $\Phi(\mathcal{G}, \rho)$ can be computed in $O(\log n)$ time.*

4.2. The general case. Theorems 3.9 and 4.3 represent roughly two extremes of the spectrum, because we need at least $O(n)$ space, and we cannot hope to answer a query in $o(\log n)$ time. The general case where the allowed storage m assumes an intermediate value between $n \log^3 n$ and n^2 is handled as follows. For technical reasons we assume for the time being that no endpoint of a segment in $\mathcal{G}$ has degree > 3 (that is, incident to more than three segments of $\mathcal{G}$). In §4.3 we show how to handle degenerate cases (i.e., when there are endpoints of degree > 3).

Using the algorithm of [Agb], partition the dual plane, in time $O(nr \log n \log^{\omega - 1} r)$, into $M = O(r^2)$ triangles $\triangle_1, \cdots, \triangle_M$, each meeting at most $\frac{n}{r}$ lines dual to the endpoints of the given segments, where r is a parameter to be chosen later. Let $\mathcal{L}_i^*$ denote the set of dual lines that intersect the triangle $\triangle_i$ for $i = 1, \cdots, M$; $|\mathcal{L}_i^*| \leq \frac{n}{r}$. For each $\triangle_i$, define the subset $\mathcal{G}_i$ of $\mathcal{G}$ to consist of all segments e having at least one endpoint whose dual is in $\mathcal{L}_i^*$. Obviously $|\mathcal{G}_i| \leq \frac{3n}{r}$. We define $W_i \subset \mathcal{G}$ as

$$W_i \; = \; \{ e \mid (e \in \mathcal{G}) \wedge (\triangle_i \subset e^*) \}.$$

LEMMA 4.4. *For each point p lying inside $\triangle_i$, the line p^* does not intersect any segment of $\mathcal{G} - (\mathcal{G}_i \cup W_i)$.*

Lemma 4.4 implies that

$$\Phi(\mathcal{G}, \rho) \; = \; \min\{\Phi(W_i, \rho),\, \Phi(\mathcal{G}_i, \rho)\}, \tag{4.1}$$

where $\triangle_i$ is the triangle containing the image of ρ. Using the same argument as in Lemma 4.2, we can prove

LEMMA 4.5. *All lines whose dual points lie inside $\triangle_i$ intersect the segments of W_i in the same order.*

We can thus order the segments of W_i in the order provided by Lemma 4.5, and compute $\Phi(W_i, \rho)$, for any ray ρ whose image lies in $\triangle_i$, in $O(\log n)$ time, using binary search. Let $\triangle_1$ and $\triangle_2$ be two adjacent triangles and let

$$\mathcal{G}_{12} \; = \; \{ e \mid (e \in \mathcal{G}_1) \wedge (\triangle_2 \subset e^*) \}. \tag{4.2}$$

It follows from the definition of W_i that $W_2 = (W_1 \cup \mathcal{G}_{12}) - \mathcal{G}_2$. Since $|\mathcal{G}_1|, |\mathcal{G}_2| = O(\frac{n}{r})$, we have $|W_2 - W_1| = O(\frac{n}{r})$. As earlier, we define a graph $\mathcal{D}$, whose vertices are the triangles Δ_i and whose edges connect pairs of vertices representing adjacent triangles.

Now an edge between v_1 and v_2 has the set $\mathcal{G}_1 \cup \mathcal{G}_2$ associated with it. Again, we construct a path Π on a spanning tree of $\mathcal{D}$, and obtain a persistent data structure $\Upsilon_1(\mathcal{G})$ to store W_i for all triangles. It can be easily shown that $\Upsilon_1(\mathcal{G})$ requires $O(nr)$ space, and can be constructed in $O(nr \log n)$ time. For any ray ρ, $\Phi(W_i, \rho)$ can still be computed in $O(\log n)$ time, where $\triangle_i$ is the triangle containing the image of ρ.

We preprocess each $\mathcal{G}_i$ into a data structure $\Upsilon_2(\mathcal{G}_i)$ of size $O(|\mathcal{G}_i| \log^3 |\mathcal{G}_i|)$ for ray shooting queries, as described in §3, so that for any ray having its image in $\triangle_i$, we can find $\Phi(\mathcal{G}_i, \rho)$ in $O(\sqrt{\frac{n}{r}} \log^2 \frac{n}{r})$ time.

To compute $\Phi(\mathcal{G}, \rho)$, for a given ray ρ, we first find the triangle $\triangle_i$ that contains its image; this can be done in $O(\log n)$ time, using an efficient point location algorithm [EGS], [ST]. It follows from (4.1) that $\Phi(\mathcal{G}, \rho)$ can be computed by calculating $\Phi(W_i, \rho)$ and $\Phi(\mathcal{G}_i, \rho)$, as described above; therefore the query time is

$$Q(n) = O\left(\sqrt{\frac{n}{r}} \log^2 \frac{n}{r} + \log n\right).$$

As for the space complexity $S(n)$, we will need $O(r^2)$ space to store the triangle $\triangle_1, \cdots, \triangle_M$, $O(nr)$ space to store Υ_1 and $O(\frac{n}{r} \log^3 \frac{n}{r})$ to store each $\mathcal{G}_i$ (cf. Theorem 3.9). Thus,

$$S(n) = O(nr) + O\left(r^2 \cdot \frac{n}{r} \log^3 \frac{n}{r}\right) = O\left(nr \log^3 \frac{n}{r}\right).$$

If we choose $r = m/n \log^3 \frac{n}{\sqrt{m}}$, then $S(n) = O(m)$ and the query time becomes

$$\begin{aligned} Q(n) &= O\left(\sqrt{\frac{n}{m/(n \log^3 \frac{n}{\sqrt{m}})}} \cdot \log^2 \frac{n}{\sqrt{m}} + \log n\right) \\ &= O\left(\frac{n}{\sqrt{m}} \log^{7/2} \frac{n}{\sqrt{m}} + \log n\right). \end{aligned}$$

Next, we bound the preprocessing time $P(n)$. We can compute $\triangle_1, \cdots, \triangle_M$ in $O(nr \log n \cdot \log^{\omega-1} r)$ time (see [Agb]). Since Υ_1 can be constructed in $O(nr \log n)$ time and each $\mathcal{G}_i$ can be preprocessed in $O((\frac{n}{r})^{3/2} \log^\omega \frac{n}{r})$ time (cf. Theorem 3.9), we have

$$\begin{aligned} P(n) &= O(nr \log n \log^{\omega-1} r) + O\left(r^2 \cdot \frac{n^{3/2}}{r^{3/2}} \log^\omega \frac{n}{r}\right) \\ &= O\left(nr \log n \log^{\omega-1} r + n^{3/2} \sqrt{r} \log^\omega \frac{n}{r}\right) \\ &= O\left(n \frac{m}{n \log^3 \frac{n}{\sqrt{m}}} \log n \log^{\omega-1} \frac{m}{n} + n^{3/2} \sqrt{\frac{m}{n \log^3 \frac{n}{\sqrt{m}}}} \log^\omega \frac{n}{\sqrt{m}}\right) \\ &= O\left(m \log^\omega n + n\sqrt{m} \log^{\omega-3/2} \frac{n}{\sqrt{m}}\right). \end{aligned}$$

Since we need $O(nr)$ space to compute $\triangle_1, \cdots, \triangle_M$, and $O(\frac{n^{3/2}}{r^{3/2}})$ to preprocess each $\mathcal{G}_i$, the total space required for preprocessing is

$$O\left(nr + \frac{n^{3/2}}{r^{3/2}}\right) = O\left(\frac{m}{\log^3 \frac{n}{\sqrt{m}}} + \frac{n^{3/2}}{m^{3/2}/(n \log^3 \frac{n}{\sqrt{m}})^{3/2}}\right)$$

$$= O\left(m + \frac{n^3}{m^{3/2}} \log^{9/2} \frac{n}{\sqrt{m}}\right).$$

Hence, we can conclude the following theorem.

THEOREM 4.6. *Given a collection $\mathcal{G}$ of n nonintersecting segments in the plane with the property that no endpoint has degree > 3, and a parameter $n \log^3 n < m < n^2$, we can preprocess $\mathcal{G}$, in $O(m \log^{\omega} n + n\sqrt{m} \log^{\omega - 3/2} n)$ time, into a data structure of size $O(m)$ so that, for query ray ρ, we can compute $\Phi(\mathcal{G}, \rho)$ in $O((n/\sqrt{m}) \log^{7/2}(n/\sqrt{m}) + \log n)$ time. The working storage required for preprocessing is $O(m + (n^3/m^{3/2}) \log^{9/2}(n/\sqrt{m}))$.*

Remark 4.7.

(i) If we allow randomization, then using Matoušek's algorithm $\mathcal{G}_i$ can be preprocessed in $O\left(\left(\frac{n}{r}\right)^{4/3} \log^2 n\right)$ time, but the query time increases by a factor of $O(\log n)$. Therefore, following the same analysis we obtain

$$P(n) = O\left(m^{2/3}n^{2/3} + m \log^{\omega} n\right)$$

$$Q(n) = O\left(\frac{n}{\sqrt{m}} \log^{9/2} \frac{n}{\sqrt{m}} + \log n\right).$$

(ii) If we maintain a single tree data structure for each $\mathcal{G}_i$, the query time can be reduced to $O(\frac{n}{\sqrt{m}} \log^3 n)$, but the preprocessing time increases considerably.

4.3. Coping with degenerate cases. The analysis of the algorithm described in the previous subsection breaks down if the segments of $\mathcal{G}$ have endpoints of arbitrarily large degree, because then we cannot guarantee that $|\mathcal{G}_i| = O(\frac{n}{r})$, and the analysis to bound the total space required to store Υ_1 relies heavily on this bound for $|\mathcal{G}_i|$. In this subsection we overcome this difficulty by showing that, given a set $\mathcal{G}$ of n nonintersecting segments, we can transform it into another set $\mathcal{G}'$ of at most $3n$ (nonintersecting) segments such that no endpoint of a segment in $\mathcal{G}'$ has degree > 3, and $\Phi(\mathcal{G}, \rho)$ can be determined from $\Phi(\mathcal{G}', \rho)$ in $O(1)$ time.

Let $\mathcal{G}_p = \{e_1, \cdots, e_t\}$ be a subset of segments of $\mathcal{G}$ all having a common endpoint p. Let δ be the minimum distance from p to its closest neighbor in $\mathcal{G} - \mathcal{G}_p$, and let c be the circle of radius $\frac{\delta}{2}$ with p as center. For a segment $e_i \in \mathcal{G}_p$, let q_i denote the intersection point of c and e_i. Assume that the segments of $\mathcal{G}_p$ are ordered in counterclockwise direction along p. There are two cases to consider:

(i) There exist two consecutive segments in $\mathcal{G}_p$, say e_t and e_1, such that the angle between e_t and e_1 is $> 180°$. For $1 < i < t$, we remove the portion of e_i that lies in the interior of c (i.e., $\overline{pq_i}$), and add the segments $\overline{q_1q_2}, \cdots, \overline{q_{t-1}q_t}$ to $\mathcal{G}$ (see Fig. 10(a)).

(ii) The angle between every two consecutive segments of $\mathcal{G}_p$ is $< 180°$. For each $e \in \mathcal{G}_p$, we remove the portion of e that lies in the interior of c, and add the segments $\overline{q_1q_2}, \cdots, \overline{q_{t-1}q_t}, \overline{q_tq_1}$ to $\mathcal{G}$ (see Fig. 10(b)).

We repeat this process for each endpoint of the segments of $\mathcal{G}$ whose degree is greater than 3. Let $\mathcal{G}'$ be the new set of segments; obviously $|\mathcal{G}'| \leq 3n$, and each endpoint has degree ≤ 3. If the ray origin s of ρ lies inside one of the newly created little polygons, say in $\triangle pq_{i-1}q_i$ of the polygon created around the endpoint p, then $\Phi(\mathcal{G}, \rho)$ lies on one of the segments incident to q_{i-1}, q_i, and can be determined in $O(\log n)$ time by locating s in $\mathcal{A}(\mathcal{G}')$. On the other hand, if s does not lie in any of the newly created polygons and $\Phi(\mathcal{G}', \rho)$ lies on a segment of $\mathcal{G}$, then $\Phi(\mathcal{G}, \rho) = \Phi(\mathcal{G}', \rho)$. Finally, if s lies outside all newly created polygons but $\Phi(\mathcal{G}', \rho)$ lies on a segment $\overline{q_{i-1}q_i}$ and e_{i-1} (respectively, e_i)

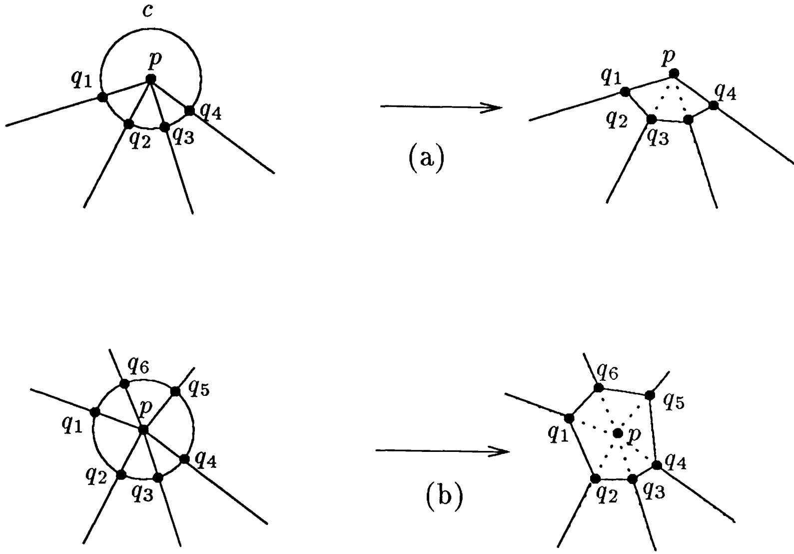


FIG. 10. *Modifying segments having a common endpoint of degree* > 3.

is the segment of $\mathcal{G}$ incident to q_{i-1} (respectively, q_i), then $\Phi(\mathcal{G}, \rho)$ lies on either e_{i-1} or e_i. Therefore, $\Phi(\mathcal{G}, \rho)$ can be computed from $\Phi(\mathcal{G}', \rho)$ in $O(1)$ time. Moreover, for each endpoint p, the minimum distance δ_p can be computed in $O(n \log n)$ time by constructing the closest point Voronoi diagram of $\mathcal{G}$ [Ya]. Hence by Theorem 4.6, we have the following theorem.

THEOREM 4.8. *Given a set $\mathcal{G}$ of n segments and a parameter $n \log^3 n < m < n^2$, we can preprocess it, in time $O(m \log^{\omega} n + n\sqrt{m} \log^{\omega - 3/2}(n/\sqrt{m}))$, into a data structure of size $O(m)$ so that, for a query ray ρ, we can compute $\Phi(\mathcal{G}, \rho)$ in $O((n/\sqrt{m}) \log^{7/2}(n/\sqrt{m}) + \log n)$ time.*

5. Reporting all intersections. In the last two sections we gave algorithms to compute $\Phi(\mathcal{G}, \rho)$ for a collection of nonintersecting segments. We now extend these algorithms to solve the following problem:

> *Given a set $\mathcal{G}$ of n nonintersecting segments, preprocess it so that, for a query ray ρ, we can quickly report all intersections $\mathcal{I}_\rho$ between $\mathcal{G}$ and ρ in their order along ρ.*

Dobkin and Edelsbrunner [DE] have given an algorithm that preprocesses $\mathcal{G}$ into a data structure of linear size so that, for a query ray ρ, $\mathcal{I}_\rho$ can be computed in $O(n^{0.695} + |\mathcal{I}_\rho|)$ time. (In fact, their algorithm works for an arbitrary collection of segments.) We first present an algorithm that uses roughly linear space, by generalizing the algorithm described in §3.

Preprocess $\mathcal{G}$, as in §3, in $O(n^{3/2} \log^{\omega} n)$ time using $O(n \log^3 n)$ space. For a given ray ρ, we compute $\mathcal{I}_\rho$ as follows. Let ℓ denote the line containing the ray ρ, and let $\mathcal{C}$ be a spanning path in C that intersects ℓ in $O(\sqrt{n})$ edges. As described in §2, compute $V_{\mathcal{B}}(\ell)$ in $O(\sqrt{n} \log n)$ time. Now we report all intersection points in $\mathcal{I}_p$ by walking along the ray ρ and stopping at each point of $\mathcal{I}_\rho$. For a point $q \in \rho$, let ρ_q be the ray emanating from q and contained in ρ.

The algorithm maintains the following invariant: When we are at a point $q \in \rho$, we maintain a list of all points $\Phi(\mathcal{G}_v, \rho_q)$ for $v \in V_{\mathcal{B}}(\ell)$, as a priority queue $\mathcal{Q}$ (with respect to

their order along ρ). Observe that $\mathcal{Q}$ remains the same between two consecutive points of $\mathcal{I}_\rho$ and that the root of $\mathcal{Q}$ stores the point of $\mathcal{I}_\rho$ that we are going to encounter next. Therefore, it suffices to show how to update $\mathcal{Q}$ after visiting a point of $\mathcal{I}_\rho$. Suppose, when we are at a point q, the root of $\mathcal{Q}$ stores $\sigma = \Phi(\mathcal{G}_u, \rho_q)$ for some $u \in V_{\mathcal{B}}(\ell)$. It is easily seen that when we cross σ, the next intersection point of ρ and $\mathcal{G}_v$, for $v \in V_{\mathcal{B}}(\ell) - \{u\}$ does not change. Thus $\mathcal{Q}$ can be updated by deleting σ from $\mathcal{Q}$ and inserting $\Phi(\mathcal{G}_u, \rho_\sigma)$ in $\mathcal{Q}$ provided $\Phi(\mathcal{G}_u, \rho_\sigma) \neq \infty$. Continue this process until $\mathcal{Q}$ becomes empty. It is easily seen that this procedure reports all intersection points of ρ and the segments of $\mathcal{G}$ in their order along ρ.

In order to bound the running time of the algorithm, observe that initially we spend $O(\sqrt{n}\log^2 n)$ time to construct the queue $\mathcal{Q}$ for $q = p$, and then spend $O(\log n)$ time in updating $\mathcal{Q}$ after each intersection. Hence, we have the following theorem.

THEOREM 5.1. *Given a collection $\mathcal{G}$ of n nonintersecting segments, we can preprocess it, in time $O(n^{3/2}\log^\omega n)$, into a data structure of size $O(n\log^3 n)$ so that, given a query ray ρ, $\mathcal{I}_\rho$ can be computed in $O(\sqrt{n}\log^2 n + |\mathcal{I}_\rho|\log n)$ time.*

An immediate corollary of Theorem 5.1 is the following.

COROLLARY 5.2. *Given a collection $\mathcal{G}$ of n nonintersecting segments, we can preprocess it, in time $O(n^{3/2}\log^\omega n)$, into a data structure of size $O(n\log^3 n)$ so that, given a query segment e, we can compute all K intersections between e and $\mathcal{G}$ in time $O(\sqrt{n}\log^2 n + K\log n)$.*

Next we show that, as in §4, the query time can be improved if we allow more space. Now preprocess $\mathcal{G}$ as described in §4 (if the segments of $\mathcal{G}$ have endpoints with degree > 3, we modify the set $\mathcal{G}$, as described in §4.3). Recall that in §4 we maintain two data structures: (i) the persistent data structure Υ_1 to store W_i for each triangle $\triangle_i$, and (ii) $\Upsilon_2(\mathcal{G}_i)$ for ray shooting queries. For a query ray ρ, we compute $\mathcal{I}_\rho$ as follows.

Suppose the ray origin p lies in the triangle $\triangle_i$; let the sorted W_i be $(e_1, e_2, \cdots, e_m)$ and suppose $\Phi(W_i, \rho) \in e_k$. Then by Lemma 4.2, $e_k, \cdots, e_m$ intersect ρ in that order along ρ, and we thus obtain all intersections between W_i and ρ. The intersections between $\mathcal{G}_i$ and ρ are obtained by the procedure described above, except that the the size of $\mathcal{Q}$ is now only $O(\sqrt{\frac{n}{r}}\log\frac{n}{r})$ because $|\mathcal{G}_i| \leq \frac{n}{r}$. $\mathcal{I}_\rho$ is then obtained by merging the two output lists of intersections with W_i and $\mathcal{G}_i$. Hence, following the same analysis as in §4, we can conclude the following theorem.

THEOREM 5.3. *Given a collection $\mathcal{G}$ of nonintersecting segments and a parameter $n\log^3 n < m < n^2$, we can preprocess it, in time $O(m\log^\omega n + n\sqrt{m}\log^{\omega-3/2}(n/\sqrt{m}))$, into a data structure of size $O(m)$ so that, given a query ray ρ, $\mathcal{I}_\rho$ can be computed in $O((n/\sqrt{m})\log^{7/2}(n/\sqrt{m}) + \log n + |\mathcal{I}_\rho|\log(n/\sqrt{m}))$ time.*

COROLLARY 5.4. *Given a collection $\mathcal{G}$ of n nonintersecting segments and a parameter $n\log^3 n < m < n^2$, we can preprocess it, in $O(m\log^\omega n + n\sqrt{m}\log^{\omega-3/2}(n/\sqrt{m}))$ time, into a data structure of size $O(m)$ so that, given a segment e, we can compute all K intersections between e and $\mathcal{G}$ in $O((n/\sqrt{m})\log^{7/2}(n/\sqrt{m}) + \log n + K\log(n/\sqrt{m}))$ time.*

6. Ray shooting in general arrangements of segments. In this section we extend our algorithm to arrangements of possibly intersecting segments. The section is organized as follows. In §6.1 we describe how to preprocess $\mathcal{G}$ for ray shooting queries, and in §6.2 we show how to answer a query. We analyze the time and space complexity of our algorithm in §6.3 and finally, in §6.4, we derive a trade-off between space and query time, similar to that of §4.

6.1. Preprocessing the segments. In this section $\mathcal{G}$ denotes an arbitrary collection of n segments in the plane. To simplify the exposition, we assume that the segments of

$\mathcal{G}$ are bounded. The preprocessing of $\mathcal{G}$ is done as follows. We construct a partition tree $\mathcal{T}$, and associate with each node $v \in \mathcal{T}$ a collection $\mathcal{G}_v \subseteq \mathcal{G}$ of n_v segments, a triangle $\triangle_v$, and another auxiliary set $\mathcal{G}'_v$ of n'_v segments. If $n_v \leq c$, for some fixed constant c, then v is a leaf of $\mathcal{T}$. Otherwise it is an internal node of $\mathcal{T}$, which is further processed as follows. For some fixed constant $r \geq 2$, partition $\triangle_v$ into $M = O(r^2)$ triangles $\triangle_1, \cdots, \triangle_M$, using the algorithm of [Agb] (or of [Maa]), so that the interior of each triangle $\triangle_i$ intersects at most $\frac{n_v}{r}$ lines containing the segments of $\mathcal{G}_v$. Create M children $w_1, \cdots, w_M$ of v, and associate with each child w_i the corresponding triangle $\triangle_{w_i} = \triangle_i$. We put a segment $e \in \mathcal{G}_v$ in $\mathcal{G}_{w_i}$ if at least one of the endpoints of e lies in $\triangle_i$. We also associate with w_i an auxiliary set $\mathcal{G}'_{w_i}$ of all segments of $\mathcal{G}_v$ that intersect the interior of $\triangle_i$. For the sake of convenience we regard each element of $\mathcal{G}'_{w_i}$ as the subsegment $e \cap \triangle_{w_i}$ of the corresponding segment e. Let $\mathcal{M}_v$ be the planar map formed by the triangles $\triangle_1, \cdots, \triangle_M$. The root u of $\mathcal{T}$ is associated with $\mathcal{G}$ itself, and $\triangle_u$ is a triangle that contains all the segments of $\mathcal{G}$. Moreover, $\mathcal{G}'_u = \mathcal{G}$, by definition.

We preprocess each node $v \in \mathcal{T}$ as follows. Preprocess the planar map $\mathcal{M}_v$ for point location queries (see [EGS] and [ST]) and store the resulting data structure at v. Let $\mathcal{L}_v$ denote the set of lines containing the segments of $\mathcal{G}'_v - \mathcal{G}_v$; $|\mathcal{L}_v| \leq n'_v$. Preprocess $\mathcal{L}_v$ into a data structure $\Upsilon_1(\mathcal{L}_v)$ for computing $\Phi(\mathcal{L}_v, \rho)$ as described in Edelsbrunner et al. [EGH*] (see also [Agc]). If $\Phi(\mathcal{L}_v, \rho)$ lies outside $\triangle_v$, then we reset it to $+\infty$.

DEFINITION 6.1. The *zone* of a triangle $\triangle$ in an arrangement $\mathcal{A}(\mathcal{G})$ of a set $\mathcal{G}$ of segments is the collection of the face portions $f \cap \triangle$, for all faces $f \in \mathcal{A}(\mathcal{G})$, that intersect $\partial\triangle$ (see Fig. 11).

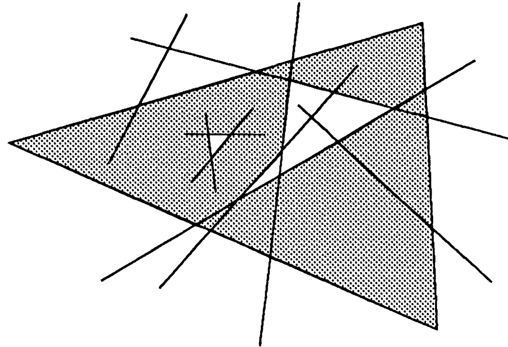


FIG. 11. *Zone of a triangle* $\triangle_i$; *some faces are nonsimple polygons.*

Using the same argument as in [EGP*], it can be proved that the total number of edges in the zone of a triangle in an arrangement of n segments is $O(n\alpha(n))$, where $\alpha(n)$ is a functional inverse of Ackermann's function. Let $\mathcal{H}_v$ denote the zone of $\triangle_v$ in $\mathcal{A}(\mathcal{G}'_v)$; $\mathcal{H}_v$ has $O(n'_v\alpha(n'_v))$ edges. $\mathcal{H}_v$ can be computed using the algorithm of Guibas et al. [GSS] because under the assumption that the segments of $\mathcal{G}'_v$ have been clipped to $\triangle_v$, $\mathcal{H}_v$ is the unbounded face of $\mathcal{A}(\mathcal{G}'_v)$. Since the edges of $\mathcal{H}_v$ are nonintersecting, we preprocess $\mathcal{H}_v$ into a data structure $\Upsilon_2(\mathcal{G}'_v)$ for computing $\Phi(\mathcal{H}_v, \rho)$, using the algorithm described in §3.

We repeat this preprocessing for every node v of $\mathcal{T}$. The resulting collection of data structures is the output of the preprocessing stage.

6.2. Answering a query. Let ρ be a query ray emanating from a point p in direction d. The query is answered by traversing a path Π_ρ of $\mathcal{T}$ and computing $\sigma_v = \Phi(\mathcal{G}_v, \rho)$ at

each node $v \in \Pi_\rho$ in a bottom-up fashion. At the end of this process we obtain at the root u, $\sigma_u = \Phi(\mathcal{G}_u, \rho) = \Phi(\mathcal{G}, \rho)$.

The path Π_ρ is defined so that for each node v along Π_ρ the ray origin p lies in $\triangle_v$. At each node $v \in \Pi_\rho$ we compute σ_v as follows. Let W_ρ be the set of children w of v for which ρ intersects $\triangle_w$. Obviously,

$$\sigma_v = \min_{w \in W_\rho} \{\Phi(\mathcal{G}'_w, \rho)\}. \tag{6.1}$$

If $w \in \Pi_\rho$, that is, p lies in $\triangle_w$, then we have already computed $\Phi(\mathcal{G}_w, \rho)$. Concerning $\sigma'_w = \Phi(\mathcal{G}'_w - \mathcal{G}_w, \rho)$, since the segments of $\mathcal{G}'_w - \mathcal{G}_w$ completely cross $\triangle_w$, σ'_w is the same as $\Phi(\mathcal{L}_w, \rho)$ (we are assuming that $\Phi(\mathcal{L}_w, \rho)$ and σ'_v are set to $+\infty$ if they do not lie in the interior of $\triangle_w$). Thus σ'_w can be computed using $\Upsilon_1(\mathcal{L}_w)$.

For all other children $z \in W_\rho$, the fact that p lies outside $\triangle_z$ implies that $\Phi(\mathcal{G}'_z, \rho)$ is the same as $\Phi(\mathcal{H}_z, \rho)$, and therefore it can be computed using $\Upsilon_2(\mathcal{G}'_z)$ (see Fig. 12).

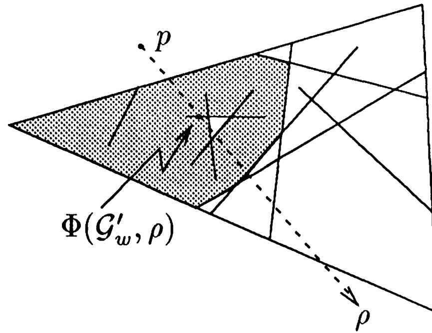


FIG. 12. $\mathcal{A}(\mathcal{G}'_w)$; *shaded region is a face of* $\mathcal{H}_w$.

Repeating the above step for all $w \in W_p$ and using (6.1) we can compute σ_v.

6.3. Analysis of the algorithm. The correctness of the algorithm follows from the above discussion, so we only have to analyze the time and space complexity of the algorithm. First, consider the query time $Q(n)$. Let Π_ρ be the path followed by the algorithm as it computes $\Phi(\mathcal{G}, \rho)$. We bound the time spent at each node $v \in \Pi_\rho$. We spend $O(\log r)$ time to find the triangle $\triangle_w$ containing the ray origin p. It follows from [EGH*] (see also [Aga] and [Agc]) that $\Phi(\mathcal{L}_w, \rho)$ can be computed in $O(\sqrt{n'_w} \log n'_w)$ time. At other triangles $\triangle_z$ intersected by ρ, we spend $O(\sqrt{n'_z \alpha(n'_z)} \log^2 n'_z)$ time to compute $\Phi(\mathcal{G}'_z, \rho)$, for $\mathcal{H}_z$ has at most $O(n'_z \alpha(n'_z))$ edge (cf. Theorem 3.9). Since $n'_z \le n_v$ for all children of v, the time spent at v is $O(r^2 \sqrt{n_v \alpha(n_v)} \log^2 n_v)$. Summing over all nodes of Π_ρ and using the fact that $r = O(1)$, we obtain

$$Q(n) = \sum_{v \in \Pi_\rho} O\Big(\sqrt{n_v \alpha(n_v)} \log^2 n\Big). \tag{6.2}$$

For a node v at level i, $n_v \le (n/r^i)$; therefore,

$$\begin{aligned} Q(n) &= \sum_{i=0}^{\log n} O\Big(\sqrt{\frac{n\alpha(n)}{r^i}} \log^2 n\Big) \\ &= O(\sqrt{n\alpha(n)} \log^2 n), \end{aligned}$$

because $r \geq 2$. Next, let us analyze the space complexity $S(n)$ and the preprocessing time $P(n)$ of our algorithm. At each node $v \in \mathcal{T}$ we store the following data structures:

(i) $\mathcal{M}_v$: The node v is partitioned into $O(r^2)$ triangles in $O(n_v r \log n_v \log^{\omega-1} r)$ time [Agb], therefore by [EGS], $\mathcal{M}_v$ can be preprocessed, in time $O(r^2 \log r)$, into a data structure of size $O(r^2)$ for point location queries. Since r is chosen to be constant, the time bound is just $O(n_v \log n_v)$ and the space required is $O(n_v)$.

(ii) $\Upsilon_1(\mathcal{L}_v)$: It follows from the result of Edelsbrunner et al. [EGH*] (see also [Agc]) that $\Upsilon_1(\mathcal{L}_v)$ requires $O(n'_v \log^2 n'_v)$ space and $O(n'^{3/2}_v \log^\omega n'_v)$ preprocessing time.

(iii) $\Upsilon_2(\mathcal{G}'_v)$: Since $\mathcal{H}_v$ has $O(n'_v \alpha(n'_v))$ edges, $\Upsilon_2(\mathcal{G}'_v)$ requires $O(n'_v \alpha(n'_v) \log^3 n'_v)$ space and $O(n'^{3/2}_v \alpha^{3/2}(n'_v) \log^\omega n'_v)$ preprocessing time (which subsumes the time $O(n'_v \alpha(n'_v) \log^2 n'_v)$ needed to compute $\mathcal{H}_v$ [GSS]).

Thus, the space used at v is $O(n'_v \alpha(n_v) \log^3 n_v)$. Summing over all nodes of $\mathcal{T}$, we get

$$S(n) = \sum_{v \in \mathcal{T}} (n'_v \alpha(n'_v) \log^3 n'_v).$$

Observe that each triangle of $\mathcal{M}_v$ intersects $O(\frac{n_v}{r})$ segments of $\mathcal{G}_v$, so $\sum_{w \in W_\rho} n'_w = O(n_v r)$. Consequently,

$$\begin{aligned} S(n) &= \sum_{v \in \mathcal{T}} O(n_v r \alpha(n_v) \log^3 n_v) \\ &= \sum_{v \in \mathcal{T}} O(n_v \alpha(n_v) \log^3 n_v) \end{aligned}$$

Since each endpoint of a segment $e \in \mathcal{G}$ falls in the interior of only one triangle, $\triangle_v$, for each level of $\mathcal{T}$, e appears in $\mathcal{G}_v$ of at most two nodes of the same level. Let $l(v)$ denote the level of the node v in $\mathcal{T}$. Then for every $i \leq \log n$ we have

$$\sum_{l(v)=i} n_v \leq 2n. \tag{6.3}$$

Hence,

$$S(n) = \sum_{i=1}^{\log n} O(n\alpha(n) \log^3 n) = O(n\alpha(n) \log^4 n).$$

Finally, we bound the preprocessing time $P(n)$ of our algorithm. The above discussion implies that the total time spent in preprocessing is at most

$$\begin{aligned} P(n) &= \sum_{v \in \mathcal{T}} O(n'^{\,3/2}_v \alpha^{3/2}(n'_v) \log^\omega n) \\ &= \sum_{v \in \mathcal{T}} O(n_v^{3/2} \alpha^{3/2}(n_v) \log^\omega n). \end{aligned} \tag{6.4}$$

Since ω is a constant less than 4.33, we can write $\alpha^{3/2}(n_v) \log^\omega n_v$ in (6.4) as $\log^\omega n$, where ω is a different constant but whose value is still less than 4.33. Thus

$$P(n) = \sum_{i=1}^{\log n} \sum_{l(v)=i} O(n_v^{3/2} \log^\omega n)$$

$$= \sum_{i=1}^{\log n} O\left(r^i \left(\frac{n}{r^i}\right)^{3/2} \log^\omega n\right)$$
$$= O\left(n^{3/2} \log^\omega n \sum_{i=1}^{\log n} \frac{1}{(\sqrt{r})^i}\right)$$
$$\leq O(n^{3/2} \log^\omega n),$$

because $r \geq 2$. Hence, we can conclude the following theorem.

THEOREM 6.2. *Given a collection $\mathcal{G}$ of n (possibly intersecting) segments, we can preprocess $\mathcal{G}$, in time $O(n^{3/2} \log^\omega n)$, into a data structure of size $O(n\alpha(n) \log^4 n)$ so that, for any query ray ρ, we can compute $\Phi(\mathcal{G}, \rho)$ in $O(\sqrt{n\alpha(n)} \log^2 n)$ time.*

Remark 6.3. If $\mathcal{G}$ contains unbounded segments, then the triangle $\triangle_u$ associated with the root u of $\mathcal{T}$ should be a triangle that contains all intersection points and all bounded segments of $\mathcal{G}$. Such a $\triangle_u$ can be easily computed in $O(n \log n)$ time. Now for each segment $e \in \mathcal{G}$, we compute $e' = e \cap \triangle_u$ and apply our algorithm to the new set of segments. The portions of the segments lying in the exterior of $\triangle_u$ do not intersect each other, and are ordered in the nondecreasing order of their slopes along $\partial\triangle_u$ in counterclockwise direction. Therefore, if a query ray does not hit a segment of $\mathcal{G}$ inside $\triangle_u$, we can determine, in additional $O(\log n)$ time, the first segment hit by the ray outside $\triangle_u$, which shows that our algorithm works for unbounded segments as well.

6.4. Trade-off between space and query time. In this subsection we establish a trade-off between space and query time for ray shooting in general arrangements of segments. As in §4 we first give a very simple algorithm that preprocesses $\mathcal{G}$, in time $O(n^2\alpha^2(n) \log n)$, into a data structure of size $O(n^2\alpha^2(n))$ so that, given a query ray ρ, $\Phi(\mathcal{G}, \rho)$ can be computed in $O(\log n)$ time.

Compute the arrangement $\mathcal{A}(\mathcal{G})$ in time $O(n^2 \log n)$ using the line sweep method [PS] (or in time $O(n^2)$ using a more involved algorithm [EOS]), and preprocess $\mathcal{A}(\mathcal{G})$ for point location queries [EGS], [ST]. Since the edges of $\mathcal{A}(\mathcal{G})$ are nonintersecting, we can preprocess each face $f \in \mathcal{A}(\mathcal{G})$ into a data structure Υ_f for logarithmic-time ray shooting queries, using $O(|n_f|^2)$ space, where n_f is the number of edges bounding f, as described in §4.

To compute $\Phi(\mathcal{G}, \rho)$, for a query ray ρ, first locate the face f of $\mathcal{A}(\mathcal{G})$ containing the ray origin p. Obviously $\Phi(\mathcal{G}, \rho)$ lies on the boundary of f, and therefore $\Phi(\mathcal{G}, \rho) = \Phi(\partial f, \rho)$ can be computed in $O(\log n)$ time, using Υ_f. Thus, the overall query time is $O(\log n)$.

As for the storage, $\mathcal{A}(\mathcal{G})$ can be preprocessed for point location queries using $O(n^2)$ space (cf. [EGS] and [ST]). The total space required to store all Υ_f is $O(\sum_f n_f^2)$. Theorem 4.3 implies that the preprocessing time is $O(\sum_f n_f^2 \log n)$. It has been shown in [EGP*] that

$$\sum_{f \in \mathcal{A}(\mathcal{G})} n_f^2 = O(n^2\alpha^2(n)).$$

Hence, we have the following theorem.

THEOREM 6.4. *Given a collection $\mathcal{G}$ of n segments in the plane, we can preprocess $\mathcal{G}$, in time $O(n^2\alpha^2(n) \log n)$, into a data structure of size $O(n^2\alpha^2(n))$ so that, for a query ray ρ, $\Phi(\mathcal{G}, \rho)$ can be computed in $O(\log n)$ time.*

Next we give an algorithm for the general case, where $n^{1+\epsilon_0} \leq m \leq n^{2-\epsilon_1}$, for some constants $\epsilon_0, \epsilon_1 > 0$. Let $m = f(n)$. To preprocess $\mathcal{G}$ into a data structure of size $O(m)$,

we proceed in the same way as in §6.1 except that at each node $v \in \mathcal{T}$ we are allowed more space, so we construct larger-size data structures that facilitate faster ray shooting in $\mathcal{L}_v$, $\mathcal{G}'_v$, etc.

Edelsbrunner et al. [EGH*] (see also [Agc]) have shown that, given a set $\mathcal{L}$ of n lines and a parameter $1 \le \beta < n$, $\mathcal{L}$ can be preprocessed, in $O(n^{3/2}\sqrt{\beta}\log^{\omega} n)$ time,[1] into a data structure $\Upsilon_1(\mathcal{L})$ of size $O(n\beta\log^2 n)$ so that, for any query ray ρ, we can compute $\Phi(\mathcal{L}, \rho)$ in $O(\sqrt{n/\beta}\log n)$ time. At each node v of level i, we store $\Upsilon_1(\mathcal{L}_v)$ with an appropriate value of $\beta = \beta_i$ (to be specified later).

Similarly, we have shown in §4 that, given a set $\mathcal{E}$ of n nonintersecting segments and a parameter β, we can preprocess $\mathcal{E}$, in time $O(n^{3/2}\sqrt{\beta}\log^{\omega} n)$, into a data structure $\Upsilon_2(\mathcal{E})$ of size $O(n\beta\log^3 n)$ so that, for a query ray ρ, we can compute $\Phi(\mathcal{E}, \rho)$ in time $O(\sqrt{n/\beta}\log^2 n)$. For a node v at level i, we store $\Upsilon_2(\mathcal{G}'_v)$ with $\beta = \beta_i$. (Recall that if $\mathcal{H}_v$ has a vertex of degree > 3, then the segments of $\mathcal{H}_v$ need to be modified, as described in §4.3.)

Next, we analyze the complexity of this algorithm. First, consider the space used by our algorithm. Since $|\mathcal{L}_v| \le n'_v$ and $|\mathcal{H}_v| = O(n'_v\alpha(n'_v))$, the space used by a node v of $\mathcal{T}$ at level i is $O(n'_v\beta_i\log^2 n'_v + n'_v\alpha(n'_v)\beta_i\log^3 n'_v) = O(n'_v\alpha(n)\beta_i\log^3 n'_v)$. The total space used is therefore

$$\begin{aligned} S(n) &= \sum_{i=0}^{\log n}\sum_{l(v)=i} O(n'_v\alpha(n)\beta_i\log^3 n'_v) \\ &= \sum_{i=0}^{\log n} O\left(\left(\sum_{l(v)=i} n'_v\right)\beta_i\alpha(n)\log^3 n'_v\right) \\ &= \sum_{i=0}^{\log n} O(\beta_i n\alpha(n)\log^3 n_i), \end{aligned}$$

where n_i is the maximum value of $|\mathcal{G}'_v|$ for a node v at level i. The last equality follows from (6.3) and the fact that $n'_v \le rn_v$. If we choose $\beta_i = (f(n_i)/(n_i\alpha(n)\log^3 n_i))$, which is easily seen to satisfy $\beta_i > 1$, then

$$\begin{aligned} S(n) &= \sum_{i=0}^{\log n} O\left(\frac{f(n_i)}{n_i\alpha(n)\log^3 n_i} n\alpha(n)\log^3 n_i\right) \\ &= O\left(\sum_{i=0}^{\log n} r^i f(n/r^i)\right) \\ &= O(f(n)) \quad \text{because } f(n) > n^{1+\epsilon_0} \\ &= O(m). \end{aligned}$$

As for the query time,

$$Q(n) = \sum_{i=0}^{\log n} O\left(\sqrt{\frac{n_i\alpha(n_i)}{\beta_i}}\log^2 n_i\right)$$

[1] Actually, the preprocessing time is $O((n\beta + n^{3/2}\sqrt{\beta})\log^{\omega} n)$, but it can be verified that for our choice of β the first term never dominates, so for simplicity we only write the second term.

$$
\begin{aligned}
&= \sum_{i=0}^{\log n} O\left(\sqrt{\frac{n_i\alpha(n_i)}{f(n_i)/(n_i\alpha(n)\log^3 n_i)}}\log^2 n_i\right)\\
&= \sum_{i=0}^{\log n} O\left(\frac{n/r^i}{\sqrt{f(n/r^i)}}\alpha(n)\log^{7/2} n\right)\\
&= O\left(\frac{n}{\sqrt{f(n)}}\alpha(n)\log^{7/2} n\right) \quad \text{because } f(n) < n^{2-\epsilon_1}\\
&= O\left(\frac{n\alpha(n)}{\sqrt{m}}\log^{7/2} n\right).
\end{aligned}
$$

Finally, the preprocessing time at a node v of level i is $O(n'^{3/2}_v\sqrt{\beta_i}\alpha^{3/2}(n'_v)\log^\omega n'_v)$. The total preprocessing time is thus

$$
\begin{aligned}
P(n) &= \sum_{i=0}^{\log n}\sum_{l(v)=i} O(n'^{3/2}_v\sqrt{\beta_i}\alpha^{3/2}(n'_v)\log^\omega n'_v)\\
&= \sum_{i=0}^{\log n} O\left(\left(\sum_{l(v)=i} n'^{3/2}_v\right)\sqrt{\beta_i}\alpha^{3/2}(n_i)\log^\omega n_i\right)\\
&= \sum_{i=0}^{\log n} O\left(r^i\left(\frac{n}{r^i}\right)^{3/2}\sqrt{\frac{r^i f(n/r^i)}{n\alpha(n)\log^3 n}}\alpha^{3/2}(n)\log^\omega n\right)\\
&= O\left(n\alpha(n)\log^{\omega-3/2} n\sum_{i=0}^{\log n}\sqrt{f(n/r^i)}\right)\\
&= O(n\sqrt{f(n)}\alpha(n)\log^{\omega-3/2} n)\\
&= O(n\sqrt{m}\alpha(n)\log^{\omega-3/2} n).
\end{aligned}
$$

Using the same argument as for (6.5), we can ignore the term $\alpha(n)$ in the above equality. Hence, we can conclude the following theorem.

THEOREM 6.5. *Given a set $\mathcal{G}$ of n segments and a parameter $n^{1+\epsilon_0} \le m \le n^{2-\epsilon_1}$, for some constants $\epsilon_0, \epsilon_1 > 0$, we can preprocess $\mathcal{G}$, in $O(n\sqrt{m}\log^{\omega-3/2} n)$ time, into a data structure of size $O(m)$ so that, for any query ray ρ, we can compute $\Phi(\mathcal{G},\rho)$ in time $O\left((n\alpha(n)/\sqrt{m})\log^{7/2} n\right)$.*

Remark 6.6. The algorithm of [EGH*] actually constructs $\Upsilon_1(\mathcal{L})$, in time

$$O\left(n\beta\log n\log^{\omega-1}\beta + n^{3/2}\sqrt{\beta}\log^\omega\frac{n}{\beta}\right),$$

using $O(n\beta\log\frac{n}{\beta})$ space, and answers a query in time $O(\sqrt{n/\beta}\log\frac{n}{\beta})$. Similarly the algorithm described in §4 constructs $\Upsilon_2(\mathcal{G})$, in time

$$O\left(n\alpha(n)\beta\log n\log^{\omega-1}\beta + (n\alpha(n))^{3/2}\sqrt{\beta}\log^\omega\frac{n}{\beta}\right),$$

using $O(n\beta\log\frac{n}{\beta})$ space, and answers a query in $O(\sqrt{n/\beta}\log^2\frac{n}{\beta})$ time. Using these bounds in the above analysis, we can improve the query time $Q(n)$ to

$$O\left(\frac{n\alpha(n)}{\sqrt{m}}\log^{7/2}\left(\frac{n\alpha(n)}{\sqrt{m}}\right) + \log n\right).$$

The preprocessing time is now

$$O\left(m \log^{\omega} n + n\alpha(n)\sqrt{m} \log^{\omega-3/2}\left(\frac{n\alpha(n)}{\sqrt{m}}\right)\right).$$

7. Implicit point location. The planar point location problem is a well-studied problem in computational geometry [Ki], [EGS], [ST]. In this problem we are to preprocess a given planar subdivision so that, for a query point, we can quickly determine the face of the subdivision containing it. Guibas et al. [GOS] have considered a generalization of this problem, in which the map is defined as the arrangement of n possibly intersecting polygonal objects of some simple shape, and the goal is to compute, for a query point p, certain information related to its position within the arrangement of the objects; for example, to determine whether p lies in the union of the objects. For simplicity we break the given objects into a collection of segments, and consider the following formal statement of the problem:

> *We are given a collection $\mathcal{G} = \{e_1, \cdots, e_n\}$ of n segments, and with each segment e we associate a function ψ_e defined on the entire plane, which assumes values in some associative and commutative semigroup S (denote its operation by $+$). Define $\Psi(x) = \sum_{e \in \mathcal{G}} \psi_e(x)$. We want to preprocess $\mathcal{G}$ so that, for any query point p, we can quickly compute $\Psi(p)$.*

We assume that ψ_e and Ψ satisfy the following conditions:

(i) For any given point x, $\psi_e(x)$ can be computed in $O(1)$ time.

(ii) Any two values in S can be added in $O(1)$ time.

(iii) Given a set $\mathcal{G}$ of n segments in the plane, we can preprocess it in time $O(n \log^k n)$, for some constant $k \geq 0$, into a linear-size data structure $\mathcal{D}(\mathcal{G})$ so that, given a point x lying either above all the lines containing the segments of $\mathcal{G}$, or below all these lines, $\Psi(x)$ can be calculated in $O(\log n)$ time.

It is shown in [GOS] that many natural problems including the problem of determining whether p lies in the union of the given objects, or of counting how many objects contain p, fall into this scheme. See also the following section for details.

The goal is to come up with an algorithm that uses $O(n \log^{O(1)} n)$ space and computes $\Psi(p)$, for any query point p, in sublinear time. Guibas et al. [GOS] gave a randomized algorithm, with $O(n \log^{k+1} n)$ expected running time, to construct a data structure of $O(n)$ size so that, for a query point p, $\Psi(p)$ can be computed in $O(n^{2/3+\delta})$ time for any $\delta > 0$. In this section we present an algorithm that improves the query time to $O(\sqrt{n} \log^2 n)$, and makes the preprocessing deterministic (albeit no longer close to linear).

Let $\mathcal{L}$ denote the set of lines containing the segments of $\mathcal{G}$. Dualize the lines of $\mathcal{L}$ to obtain a set $\mathcal{L}^*$ of n points. Let $\mathbf{C} = \{\mathcal{C}_1, \cdots, \mathcal{C}_k\}$ denote a family of $k = O(\log n)$ spanning paths on $\mathcal{L}^*$ with $\sigma(\mathbf{C}) = O(\sqrt{n})$. We show how to preprocess a single path $\mathcal{C} \in \mathbf{C}$.

First, construct a binary tree $\mathcal{B} = \mathcal{B}(\mathcal{C})$ as in §2. With each node v of $\mathcal{B}$ we associate a set $\mathcal{G}_v$ of segments $e \in \mathcal{G}$ such that the dual of the line containing e belongs to S_v (as defined in §2). At each node v we store $\mathcal{D}(\mathcal{G}_v)$ so that, for any query point p lying either above all the lines containing the segments of $\mathcal{G}_v$ or below all of them, $\Psi_v(p) = \sum_{e \in \mathcal{G}_v} \psi_e(p)$ can be computed in $O(\log n)$ time.

For a given query point p, we compute $\Psi(p)$ as follows. Let p^* denote the dual of p. Obviously,

$$\sum_{i=1}^{n} \psi_{e_i}(p) = \sum_{v \in V_{\mathcal{B}}(p^*)} \Psi_v(p).$$

Therefore, it suffices to show how to compute $\Psi_v(p)$, for a node $v \in V_{\mathcal{B}}(p^*)$. Observe that for any $v \in V_{\mathcal{B}}(p^*)$, p^* lies either above all the points of S_v, or below all of them, say below. Since duality preserves the above-below relationship, p lies below all the lines containing the segments of $\mathcal{G}_v$. Therefore, $\Psi_v(p)$ can be easily computed in $O(\log n)$ time using $\mathcal{D}(\mathcal{G}_v)$.

Next, let us analyze the complexity of our algorithm. First consider the time spent in answering a query. By Theorem 2.1, we can determine, in $O(\log n)$ time, a path $\mathcal{C} \in \mathbf{C}$ that intersects p^* in at most $O(\sqrt{n})$ edges, and it follows from the discussion in §2 that $V_{\mathcal{B}}(p^*)$, for a given line p^*, can be computed in $O(\sqrt{n}\log n)$ time. By property (iii), for each $v \in V_{\mathcal{B}}(p^*)$, $\Psi_v(p)$ can be calculated in $O(\log n)$ time. The total time spent is thus $O(\sqrt{n}\log^2 n)$. As for the space complexity, $\mathcal{D}(\mathcal{G}_v)$ requires $O(|\mathcal{G}_v|)$ space. Since the segments associated with the nodes of $\mathcal{B}$ at the same level are pairwise disjoint, the total space required to store $\mathcal{B}$ is $O(n\log n)$. Finally, the preprocessing time is bounded by the time spent in computing $\mathbf{C}$ plus the time spent in preprocessing $\mathcal{G}_v$ for all $v \in \mathcal{B}$. Hence, the total preprocessing time is $O(n^{3/2}\log^{\omega} n + n\log^{k+2} n) = O(n^{3/2}\log^{\omega} n)$.

Therefore, we can conclude the following theorem.

THEOREM 7.1. *Given a collection $\mathcal{G}$ of n segments, and function ψ_e associated with each segment satisfying properties* (i)–(iii)*, we can preprocess $\mathcal{G}$, in $O(n^{3/2}\log^{\omega} n)$ time, into a data structure of size $O(n\log^2 n)$ so that, for any query point p, $\Psi(p)$ can be computed in $O(\sqrt{n}\log^2 n)$ time.*

Remark 7.2.

(i) As in §3, we can reduce the space complexity to $O(n\log n)$ by maintaining a single tree structure instead of a family of $O(\log n)$ trees. Also, if we allow randomization, then the (expected) preprocessing time is $O(n^{4/3}\log^2 n)$, but the query time increases by a factor of $\log n$.

(ii) In some applications, where calculation of $\Psi(x)$ in (iii) above is accomplished by a binary search, it is possible to reduce the query time to $O(\sqrt{n}\log n)$, using fractional cascading.

(iii) As in the case of the ray shooting problem, the query time can be improved by allowing more storage. Instead of describing the trade-off for the general case, we will describe it in the next section for a specific example.

(iv) In a companion paper [Agc] we solve the *batched* version of this problem, where all the query points p are given in advance. We present there a solution that runs in time

$$O\left(m^{2/3}n^{2/3}\log^{2/3} n \log^{\omega/3}\frac{n}{\sqrt{m}} + n\log^k n \log\frac{n}{\sqrt{m}} + m\log n\right),$$

where m is the number of given query points.

8. Other applications. In this section we consider other applications of our technique. All these problems were studied in [GOS], who obtained algorithms with $O(n^{2/3+\delta})$ query time, for any $\delta > 0$. We show that using our approach the query time can be reduced to roughly $\sqrt{n}$.

8.1. Polygon containment problem: Preprocessing version. First consider the following problem:

> *Given a set* $\mathbf{T}$ *of* n *(possibly intersecting) triangles, we want to preprocess* $\mathbf{T}$ *so that, given a query point* p*, we can quickly count the number of triangles in* $\mathbf{T}$ *containing* p *(or just determine whether* p *lies in the union of these triangles); see Fig.* 13.

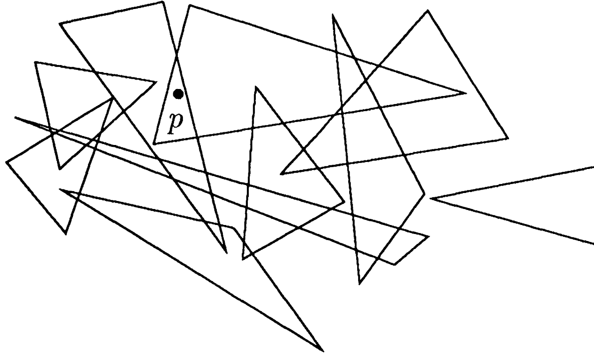


FIG. 13. *Polygon containment problem.*

We first present an algorithm that uses roughly linear space, and then show that the query time can be improved by using more space. Our algorithm is based on the following observation of [GOS]. Let $\mathcal{G}$ denote the set of edges bounding the triangles in $\mathbf{T}$ and, for each $e \in \mathcal{G}$, let $B(e)$ denote the semi-infinite trapezoidal strip lying below e. Define a function ψ_e in the plane so that $\psi_e(p) = 0$ for a point p outside B_e, and for $p \in B_e$, $\psi_e(p) = 1$, if the triangle corresponding to e lies below the line containing the segment e, otherwise $\psi_e(p) = -1$. It can be checked that $\Psi(p)$, for a point p, gives the number of triangles of $\mathbf{T}$ containing p. Moreover, ψ_e obviously satisfies properties (i) and (ii). As to property (iii), if a point p lies above all lines containing the given edges then $\Psi(p) = 0$, by definition. On the other hand, if p lies below all these lines, we do the following. Let $\hat{e}$ denote the x-projection of an edge e of some triangle. It is easily checked that

$$\Psi(p) = \sum_{p_x \in \hat{e}_j} \epsilon_j,$$

where p_x is the x-coordinate of p and ϵ_j is the nonzero value of ψ_{e_j} at p. Note that the sum of the right-hand side remains the same between two consecutive endpoints of the projected segments, and the constant values of Ψ over these intervals can be computed, in overall time $O(n \log n)$, by scanning the projected segments from left to right. Hence, we can preprocess $\mathbf{T}$, in time $O(n \log n)$, into a data structure $\mathcal{D}$ so that, for a point p lying below all lines of $\mathcal{L}$, $\Psi(p)$ can be computed in $O(\log n)$ time.

Thus, the observation of [GOS] and Theorem 7.1 imply that by preprocessing $\mathcal{G}_v$ into the above data structure $\mathcal{D}_v$, for each node v of $\mathcal{B}$, the number of triangles in $\mathbf{T}$ containing a query point p can be counted in $O(\sqrt{n} \log^2 n)$ time. But observe that each of the data structures $\mathcal{D}_v$ is a sorted list, and at each node v we do a binary search in $\mathcal{D}_v$ to compute Ψ_v. We can therefore apply the fractional cascading technique of [CGb] to the collection of lists $\mathcal{D}_v$ attached to the nodes v of $\mathcal{B}$. This will allow us to search through

the lists $\mathcal{D}_v$ of all nodes $v \in V_{\mathcal{B}}(\ell)$ in overall time $O(\log n + |V_{\mathcal{B}}(\ell)|) = O(\sqrt{n}\log n)$. Hence, we have the following theorem.

THEOREM 8.1. *Given a set* $\mathbf{T}$ *of* n *triangles in the plane, we can preprocess* $\mathbf{T}$, *in time* $O(n^{3/2}\log^{\omega} n)$, *into a data structure of size* $O(n\log^2 n)$ *so that, given a query point* p, *we can determine, in time* $O(\sqrt{n}\log n)$, *the number of triangles in* $\mathbf{T}$ *containing the point* p.

We next establish a trade-off between space and query time for the polygon containment problem. If we allow $O(n^2)$ space, then we can construct the entire arrangement $\mathcal{H}$ of $\bigcup_{e\in\mathcal{G}} B_e$. It is easily seen that the value of Ψ does not change within a face of $\mathcal{H}$, and while constructing $\mathcal{H}$ we can compute Ψ for each of its face. Now given a point p, we can compute $\Psi(p)$ in $O(\log n)$ time by locating p in $\mathcal{H}$. Thus if we allow quadratic storage, the query time can be reduced to $O(\log n)$. Next we give an algorithm for the general case when $n\log^2 n < m < n^2$.

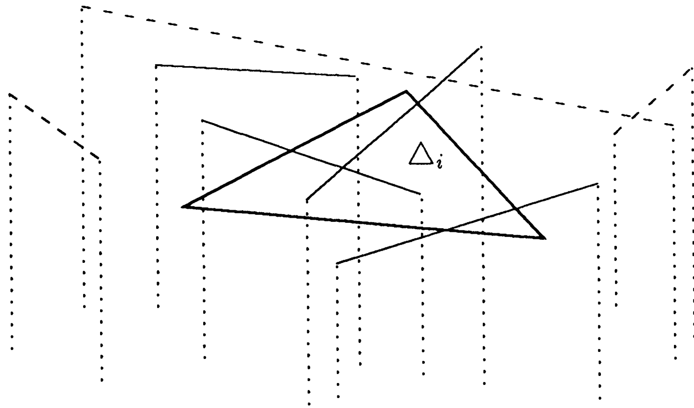


FIG. 14. *Triangle* $\triangle_i$ *and segments of* $\mathcal{G}$: *solid lines are* $\mathcal{G}_i$; *dashed lines are* $\mathcal{G}_i'$.

Let Γ denote the set of lines bounding the trapezoidal strips B_e, that is, the lines containing the segments of $\mathcal{G}$ and the vertical lines passing through the endpoints of segments in $\mathcal{G}$. Partition the plane into $M = O(r^2)$ triangles $\triangle_1, \cdots, \triangle_M$, each meeting at most $\frac{n}{r}$ lines of Γ [Agb]. With each $\triangle_i$ we associate a set $\mathcal{G}_i$ consisting of all segments $e \in \mathcal{G}$ such that either e or one of the two downward-directed vertical rays emanating from its endpoints intersects $\triangle_i$ (see Fig. 14). Let $\mathcal{G}_i' = \mathcal{G} - \mathcal{G}_i$. We can compute $\mathcal{G}_i$, for each i, in total time $O(nr\log n)$. Since $\triangle_i$ does not intersect the boundary of B_e for $e \in \mathcal{G}_i'$, ψ_e remains constant over $\triangle_i$. Moreover, $\sum_{e\in\mathcal{G}_i'}\psi_e$ for every $\triangle_i$ can be computed in $O(nr)$ time, as described in [Agc]. We preprocess $\mathcal{G}_i$ into a data structure of size $O(\frac{n}{r}\log^2\frac{n}{r})$, using the method just mentioned. For answering a query, we first locate the triangle $\triangle_k$ containing the query point p. Once we know $\triangle_k$, $\sum_{e\in\mathcal{G}_k'}\psi_e(p)$ can be determined in $O(1)$ time, and $\sum_{e\in\mathcal{G}_k}\psi_e(p)$ can be computed as described above. Since $|\mathcal{G}_i| = O(\frac{n}{r})$, the query time is

$$Q(n) = O\left(\sqrt{\frac{n}{r}}\log\frac{n}{r} + \log n\right).$$

We need $O(r^2)$ space to store the planar map formed by $\triangle_1, \cdots, \triangle_M$ and $O(\frac{n}{r}\log^2\frac{n}{r})$ to store the data structure constructed for each $\mathcal{G}_i$. Therefore, the total space used is

$$S(n) = O(r^2) + O\left(r^2\cdot\frac{n}{r}\log^2\frac{n}{r}\right)$$

$$= O\left(nr\log^2\frac{n}{r}\right).$$

If we choose $r = (m/n\log^2\frac{n}{\sqrt{m}})$, which is easily seen to satisfy $1 \le r < n$, then $S(n) = O(m)$, and the query time is

$$\begin{aligned} Q(n) &= O\left(\sqrt{\frac{n}{m/(n\log^2\frac{n}{\sqrt{m}})}}\log\frac{n}{\sqrt{m}} + \log n\right) \\ &= O\left(\frac{n}{\sqrt{m}}\log^2\frac{n}{\sqrt{m}} + \log n\right). \end{aligned}$$

Finally, the preprocessing time is

$$\begin{aligned} P(n) &= O(nr\log n\log^{\omega-1} r) + O\left(r^2\cdot\left(\frac{n}{r}\right)^{3/2}\log^{\omega}\frac{n}{r}\right) \\ &= O\left(m\log^{\omega} n + n^{3/2}\sqrt{\frac{m}{n\log^2\frac{n}{\sqrt{m}}}}\log^{\omega}\frac{n}{\sqrt{m}}\right) \\ &= O\left(m\log^{\omega} n + n\sqrt{m}\log^{\omega-1}\frac{n}{\sqrt{m}}\right). \end{aligned}$$

Hence, we can conclude the following theorem.

THEOREM 8.2. *Given a collection* $\mathbf{T}$ *of* n *(possibly intersecting) triangles in the plane, we can preprocess* $\mathbf{T}$*, in time* $O(m\log^{\omega} n + n\sqrt{m}\log^{\omega-1}\frac{n}{\sqrt{m}})$*, into a data structure of size* $O(m)$ *so that, for a query point* p*, we can count the number of triangles of* $\mathbf{T}$ *containing* p *in time* $O(\frac{n}{\sqrt{m}}\log^2\frac{n}{\sqrt{m}} + \log n)$.

Remark 8.3. The batched version of this problem, when all points are given in advance, can be solved, in time $O(m^{2/3}n^{2/3}\log^{2/3} n\log^{\omega/3}\frac{n}{\sqrt{m}} + (m+n)\log n)$, using a different technique [Agc].

8.2. Implicit hidden surface removal. The next problem that we consider is the following version of hidden surface removal problem:

Given a collection of opaque objects in three-dimensional space, and a viewing point a, we wish to calculate the scene obtained by viewing these objects from a.

The hidden surface removal problem has been extensively studied by many researchers (see, e.g., [De] and [MK]), because of its applications in graphics and other areas. For the sake of simplicity let us restrict our attention to polyhedral objects, whose boundary $\mathbf{T}$ is a collection $\{\triangle_1, \cdots, \triangle_n\}$ of n nonintersecting triangles. In the case of *implicit* hidden surface removal, we do not want to compute the scene explicitly, but only to preprocess them so as to determine quickly the object seen at any particular query pixel [CS], [GOS]. In this subsection, we consider the following special case of the implicit hidden surface removal problem. Let $\mathbf{T} = \{\triangle_1, \cdots, \triangle_n\}$ be a collection of n nonintersecting triangles such that $\triangle_i$ lies in the plane $z = c_i$, where $0 < c_1 \le c_2 \le \cdots \le c_n$ are some fixed heights. Preprocess $\mathbf{T}$ so that, given a query point p on the xy-plane, one can determine the lowest triangle $\triangle_i$ hit by the upward-directed vertical ray emanating from p.

[GOS] have given an algorithm for this problem that uses randomized processing and has $O(n^{2/3+\delta})$ query time, for any $\delta > 0$. Their algorithm first projects all triangles

on the xy-plane, and then performs a binary search through the sequence $(\triangle_1^*, \cdots, \triangle_n^*)$ of projected triangles to find the first index j such that $\triangle_j^*$ contains the query point p. Each step of the binary search tests whether p lies in the union of some contiguous block of projected triangles, using the polygon containment algorithm. Therefore, the preprocessing step consists of constructing a binary tree $\mathcal{Z}$ on $\mathbf{T}$ whose leaves store the triangles of $\mathbf{T}$ in increasing height, and each internal node w is associated with a set of triangles $\mathbf{T}_w$, stored at the leaves of the subtree rooted at w. For each node w of $\mathcal{Z}$, preprocess $\mathbf{T}_w$ for the polygon containment problem, using the algorithm described in §8.1. It now follows from the above discussion that a query can be answered by following a path π in $\mathcal{Z}$ and solving the polygon containment problem at each node of π. Hence using Theorem 8.1, we can conclude with the following theorem.

THEOREM 8.4. *The implicit hidden surface removal problem for an ordered collection of n triangles in three-dimensional space can be solved in $O(\sqrt{n}\log^2 n)$ query time, $O(n\log^3 n)$ space, and $O(n^{3/2}\log^\omega n)$ preprocessing.*

Remark 8.5.

(i) Recently several algorithms for other variants of the implicit hidden surface removal problem have been developed; see [SML], [Be].

(ii) As in the case of the polygon containment problem, the query time can be improved if we allow more space. In particular, if we allow $O(m)$ space, where $n < m < n^2$, then $Q(n) = O(\frac{n}{\sqrt{m}}\log^2\frac{n}{\sqrt{m}} + \log^2 n)$ and $P(n) = O(m\log^\omega n + n\sqrt{m}\log^{\omega-1}\frac{n}{\sqrt{m}})$.

(iii) We can easily modify our algorithm without affecting its time complexity so that the query point p lies anywhere in $\mathbb{R}^3$, rather than lying on the xy-plane. We leave it for the reader to fill in the details.

8.3. Polygon placement problem. Finally consider the following problem:

Let P be a k-gon (not necessarily simple) and let $\mathbf{\Delta} = \{\triangle_1, \cdots, \triangle_n\}$ be a set of n (possibly intersecting) triangles. Preprocess $\mathbf{\Delta}$ so that, given a (translated) placement of P, we can quickly determine whether P intersects any of the obstacles at that placement.

Such a situation arises in several applications [Cha]. A special case, in which P is convex and the triangles are non-intersecting, has been widely studied (see, e.g., [BZ], [CD], [Fo], [LS]). But the best known solution for the general case is by [GOS], who have given an algorithm with randomized preprocessing and $O((kn)^{2/3+\delta})$ query time, for any $\delta > 0$, by reducing this problem to the polygon containment problem. Using their technique, and applying Theorem 8.1, we can easily obtain the following theorem.

THEOREM 8.6. *We can preprocess $\mathbf{\Delta}$ and P, in $O((kn)^{3/2}\log^\omega kn)$ time, into a data structure of size $O(kn\log^2 kn)$ so that, given a translated placement of P, we can determine in time $O(\sqrt{kn}\log kn)$, whether P collides with the obstacles at that placement.*

Remark 8.7. The trade-off between space and query time described in §8.1 works here as well. Therefore, if we allow $O(m)$ space, where $n < m < n^2$, then $Q(n) = O(\frac{kn}{\sqrt{m}}\log^2\frac{kn}{\sqrt{m}} + \log kn)$ and $P(n) = O(m\log^\omega kn + kn\sqrt{m}\log^{\omega-1} kn)$.

9. Conclusions. In this paper we presented efficient algorithms for various problems involving collections of segments in the plane, using spanning trees with low stabbing number. Since the submission of this paper there have been a number of significant developments on these problems. We summarize some of the new results here:

(i) Matoušek has proposed an $O(n^{3/2}\log^2 n)$ algorithm to construct a single spanning tree of a set of n points in $\mathbb{R}^2$ with $O(\sqrt{n})$ stabbing number [Mac]. It

immediately improves the space complexity and the preprocessing time of all the algorithms presented here by a factor of $\log n$ and $\log^{\omega-2} n$, respectively.

(ii) Cheng and Janardan [CJ] have shown that a set of n (possibly intersecting) segments can be preprocessed into a data structure of size $O(n \log^3 n)$ so that a ray shooting query can be answered in $O(\sqrt{n} \log n)$ time. Their algorithm is based on spanning trees with low stabbing number and therefore its space complexity and preprocessing time can also be improved by incorporating Matoušek's procedure.

(iii) Using an entirely different approach, Yehuda and Fogel [BF] have designed another ray shooting algorithm for nonintersecting segments that requires $O(n \log n)$ space and supports $O(\sqrt{n} \log n)$ time queries. The preprocessing time of their algorithm is $O(n^{3/2})$. Their algorithm can be extended to intersecting segments using the approach described in §6.

(iv) Another recent development in this area is by Chazelle et al. [CEGGSS], who showed that a polygonal region with k holes can be preprocessed into a data structure of size $O(n \log n)$ so that a ray shooting query can be answered in time $O(\sqrt{k} \log n)$. The preprocessing time of their algorithm is roughly $n^{3/2}$.

(v) A drawback of all these algorithms is that unlike Guibas et al.'s algorithm [GOS] their preprocessing time is not close to linear. Agarwal and Sharir [AS] have shown that the preprocessing can be improved to $O(n^{1+\epsilon})$ without affecting the query time significantly. In particular, their algorithm preprocesses a collection of segments, in time $O(n^{1+\epsilon})$, into a data structure of size $O(n^{1+\epsilon})$, so that a ray shooting query can be answered in $O(n^{1/2+\epsilon})$ time, where ϵ is an arbitrarily small positive constant. Their algorithm relies on a recent partitioning scheme of Chazelle et al. [CSW]. It can be modified to report all k intersections between a collection of n given segments and a query segment in time $O(n^{1/2+\epsilon} + k)$.

(vi) Another shortcoming of the above algorithms is that they do not extend to arbitrary arcs (except the algorithm of [AS]). Some progress in this direction has been made by Agarwal et al. [AKO], who have developed a ray shooting algorithm for nonintersecting Jordan arcs that answers a query in time $O(\sqrt{n} \log^2 n)$ and requires $O(n \log n)$ space.

In spite of these various developments, there are several interesting open problems:

1. The most challenging open problem is to give nontrivial lower bounds for the ray shooting and the implicit point location problems. Recently Chazelle [Chb] showed that if we allow only $O(n)$ space, then a simplex range query (i.e., counting the number of points of a given set contained in a query triangle) requires $\Omega(\sqrt{n})$ time. We conjecture that similar lower bounds hold for these problems as well.
2. Mark Overmars has posed the following problem, which is a generalization of the polygon containment problem: *Given a set* $\mathbf{T}$ *of triangles, preprocess it so that, for a query segment e, one can quickly determine if e is contained in the union of triangles of* $\mathbf{T}$. It will be interesting to come up with an efficient algorithm using spanning trees of low stabbing number.
3. Finally, there remains the task of looking for other interesting problems that can be solved efficiently using the spanning trees of low stabbing number.

Acknowledgments. I thank my advisor Micha Sharir for encouraging me to work on this problem, for several valuable discussions, and for reading earlier versions of this paper which significantly improved its quality. Thanks are also due to three anonymous

referees for several helpful comments and for pointing out an error in the earlier version of the paper.

REFERENCES

[Aga] P. K. AGARWAL, *A deterministic algorithm for partitioning arrangements of lines and its applications*, Proc. 5th Annual Symposium on Computational Geometry, 1989, pp. 11–22.

[Agb] ———, *Partitioning arrangements of lines: I. An efficient deterministic algorithm*, Discrete Comput. Geom., 5 (1990), pp. 449–483.

[Agc] ———, *Partitioning arrangements of lines: II. Applications*, Discrete Comput. Geom., 5 (1990), pp. 533-573.

[AKO] P. AGARWAL, M. VAN KREVALD, AND M. OVERMARS, *Intersection queries for curved objects*, Proc. 7th Annual Symposium on Computational Geometry, 1991, pp. 41–50.

[AS] P. AGARWAL AND M. SHARIR, *Applications of a new partitioning scheme*, Proc. 2nd Workshop on Algorithms and Data Structures, 1991, pp. 379–391; Discrete Comput. Geom., to appear.

[BF] R. BAR YEHUDA AND S. FOGEL, *Good splitters with applications to ray shooting*, Proc. 2nd Canadian Conf. on Computational Geometry, 1990, pp. 81–85.

[Be] M. BERN, *Hidden surface removal for rectangles*, J. Comput. Systems Sci., 40 (1990), pp. 49–69.

[BZ] B. BHATTACHARYA AND J. ZORBAS, *Solving the two-dimensional findpath problem using a line-triangle representation of the robot*, J. Algorithms, 9 (1988), pp. 449–469.

[Cha] B. CHAZELLE, *The polygon containment problem*, in Advances in Computing Research, Vol. I: Computational Geometry, F. P. Preparata, ed., JAI Press, Greenwich, CT, 1983, pp. 1–33.

[Chb] ———, *Lower bounds on the complexity of polytope range searching*, J. Amer. Math. Soc., 2 (1989), pp. 637–666.

[Chc] ———, *Tight bounds on the stabbing number of trees in Euclidean plane*, Tech. Report CS-TR-155-58, Dept. Computer Science, Princeton University, Princeton, NJ, May 1988.

[Chd] ———, Private communication, 1989.

[CEGGSS] B. CHAZELLE, H. EDELSBRUNNER, M. GRIGNI, L. GUIBAS, M. SHARIR, AND J. SNOEYINK, *Ray shooting in polygons using geodesic triangulations*, Proc. 18th Int. Coll. on Automata, Languages and Programming, 1991.

[CGb] B. CHAZELLE AND L. GUIBAS, *Fractional cascading: I. A data structuring technique*, Algorithmica, 1 (1986), pp. 133–162.

[CGc] ———, *Fractional cascading: II. Applications*, Algorithmica, 1 (1986), pp. 163–191.

[CGa] ———, *Visibility and intersection problems in plane geometry*, Discrete Comput. Geom., 4 (1989), pp. 551–581.

[CGL] B. CHAZELLE, L. GUIBAS, AND D. T. LEE, *The power of geometric duality*, BIT, 25 (1985), pp. 76–90.

[CSW] B. CHAZELLE, M. SHARIR, AND E. WELZL, *Quasi optimal upper bounds for simplex range searching and new zone theorem*, Proc. 6th Annual Symposium on Computational Geometry, 1990, pp. 23–33.

[CW] B. CHAZELLE AND E. WELZL, *Quasi optimal range searching in spaces with finite* VC-*dimensions*, Discrete Comput. Geom., 4 (1989), pp. 467–489.

[CD] P. CHEW AND L. DRYSDALE, *Voronoi diagrams based on convex distance functions*, Proc. 1st Annual Symposium on Computational Geometry, 1985, pp. 235–244.

[CJ] S. CHENG AND R. JANARDAN, *Space-efficient ray shooting and intersection searching: Algorithms, dynamization, and applications*, Second SIAM-ACM Symposium on Discrete Algorithms, 1991, pp. 7–16.

[Cl] K. CLARKSON, *New applications of random sampling in computational geometry*, Discrete Comput. Geom., 2 (1987), pp. 195–222.

[Co] R. COLE, *Searching and storing similar lists*, J. Algorithms, 7 (1986), pp. 202–220.

[CS] R. COLE AND M. SHARIR, *Visibility problems for polyhedral terrains*, J. Symbolic Comput., 7 (1989), pp. 11–30.

[De] F. DÉVAI, *Quadratic bounds for hidden line elimination*, Proc. 2nd Annual Symposium on Computational Geometry, 1986, pp. 269–275.

[DE] D. DOBKIN AND H. EDELSBRUNNER, *Space searching for intersecting objects*, J. Algorithms, 8 (1987), pp. 348–361.

[DSST] J. DRISCOLL, N. SARNAK, D. SLEATOR, AND R. TARJAN, *Making data structures persistent*, J. Comput. Systems Sci., 38 (1989), pp. 86–124.

[Ed] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Heidelberg, 1987.

[EG] H. Edelsbrunner and L. Guibas, *Topologically sweeping an arrangement*, J. Comput. Systems Sci., 38 (1989), pp. 165–194.

[EGH*] H. Edelsbrunner, L. Guibas, J. Hershberger, R.Seidel, M. Sharir, J. Snoeyink, and E. Welzl, *Implicitly representing arrangements of lines or segments*, Discrete Comput. Geom., 4 (1989), pp. 433–466.

[EGP*] H. Edelsbrunner, L. Guibas, J. Pach, R. Pollack, R. Seidel, and M. Sharir, *Arrangements of curves in the plane—topology, combinatorics, and algorithms*, Theoret. Comput. Sci., 92 (1992), pp. 319–336.

[EGS] H. Edelsbrunner, L. Guibas, and G. Stolfi, *Optimal point location in monotone subdivisions*, SIAM J. Comput., 15 (1986), pp. 317–340.

[EOS] H. Edelsbrunner, J. O'Rourke, and R. Seidel, *Constructing arrangements of lines and hyperplanes with applications*, SIAM J. Comput., 15 (1986), pp. 341–363.

[EW] H. Edelsbrunner and E. Welzl, *Halfplanar range search in linear space and $O(n^{0.695})$ query time*, Inform. Process. Lett., 23 (1986), pp. 289–293.

[Fo] S. Fortune, *Fast algorithms for polygon containment*, Proc. 12th International Colloquium on Automata, Languages and Programming, Lecture Notes in Comput. Sci., 194, Springer-Verlag, New York, 1985, pp. 189–198.

[GHLST] L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan, *Linear time algorithms for shortest path and visibility problems*, Algorithmica, 2 (1987) pp. 209–233.

[GOS] L. Guibas, M. Overmars, and M. Sharir, *Ray shooting, implicit point location, and related queries in arrangements of segments*, Tech. Report 433, Dept. Computer Science, New York University, New York, March 1989.

[GSS] L. Guibas, M. Sharir, and S. Sifrony, *On the general motion planning problem with two degrees of freedom*, Discrete Comput. Geom., 4 (1989), pp. 491–521.

[GY] L. Guibas and F. Yao, *On translating a set of rectangles*, in Advances in Computer Research, Vol. I: Computational Geometry, F. P. Preparata, ed., JAI Press, Greenwich, CT, 1983, pp. 61–77.

[HW] D. Haussler and E. Welzl, *ϵ-nets and simplex range queries*, Discrete Comput. Geom., 2 (1987), pp. 127–151.

[Ki] D. Kirkpatrick, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.

[LS] D. Lenen and M. Sharir, *Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams*, Discrete Comput. Geom., 2 (1987), pp. 9–31.

[MK] M. McKenna, *Worst case optimal hidden surface removal*, ACM Trans. Graphics, 6 (1987), pp. 19–28.

[Maa] J. Matoušek, *Construction of ϵ-nets*, Discrete Comput. Geom., 5 (1990), pp. 427–448.

[Mab] ———, *Spanning trees with low crossing numbers*, Inform. Théoret. Appl., 25 (1991), pp. 103–123.

[Mac] ———, *More on cutting arrangements and spanning trees with low crossing number*, Tech. Report B-90-2, Department of Mathematics, Freie Universität, Berlin, February 1990.

[PS] F. Preparata and M. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, Heidelberg, 1985.

[SML] A. Schmitt, H. Muller, and W. Leister, *Ray tracing algorithm—Theory and practice*, in Theoretical Foundations of Computer Graphics and CAD, R. Earnshaw, ed., NATO ASI Series, Vol. F-40, Springer-Verlag, New York, 1988, pp. 997–1030.

[SO] S. Suri and J. O'Rourke, *Worst case optimal algorithms for constructing visibility polygons with holes*, Proc. 2nd Annual Symposium on Computational Geometry, 1986, pp. 14–23.

[ST] N. Sarnak and R. E. Tarjan, *Planar point location using persistent search trees*, Comm. ACM, 29 (1986), pp. 669–679.

[We] E. Welzl, *Partition trees for triangle counting and other range searching problems*, Proc. 4th Annual Symposium on Computational Geometry, 1988, pp. 23–33.

[Ya] C. K. Yap, *An $O(n \log n)$ algorithm for the Voronoi diagram of a set of simple curve segments*, Discrete Comput. Geom., 2 (1987), pp. 365–393.

SIAM J. COMPUT.
Vol. 21, No. 3, pp. 571–586, June 1992

©1992 Society for Industrial and Applied Mathematics
009

THE NILPOTENCY PROBLEM OF ONE-DIMENSIONAL CELLULAR AUTOMATA*

JARKKO KARI†

Abstract. The limit set of a celullar automaton consists of all the configurations of the automaton that can appear after arbitrarily long computations. It is known that the limit set is never empty—it contains at least one homogeneous configuration. A CA is called nilpotent if its limit set contains just one configuration. The present work proves that it is algorithmically undecidable whether a given one-dimensional cellular automaton is nilpotent. The proof is based on a generalization of the well-known result about the undecidability of the tiling problem of the plane. The generalization states that the tiling problem remains undecidable even if one considers only so-called NW-deterministic tile sets, that is, tile sets in which the left and upper neighbors of each tile determine the tile uniquely. The nilpotency problem is known to be undecidable for d-dimensional CA for $d \geq 2$. The result is the basis of the proof of Rice's theorem for CA limit sets, which states that every nontrivial property of limit sets is undecidable.

Key words. cellular automaton, limit set, nilpotent, decidability, tiling

AMS(MOS) subject classification. 68D20

1. Introduction.

Cellular automata are discrete and deterministic dynamical systems. They provide simple models of complex natural systems encountered in physics, biology, and other fields. Like natural systems they consist of large numbers of simple basic components that together produce the complex behaviour of the system.

A d-dimensional cellular automaton consists of an infinite d-dimensional array of identical cells. Each cell is always in one state from a finite state set. The cells alter their states synchronously on discrete time steps according to a local rule. The rule gives the new state of each cell as a function of the old states of some nearby cells, its neighbors. The array is homogeneous so that all its cells operate under the same local rule. The states of all cells in the array are described by a configuration. The local rule of the automaton specifies a global function that tells how each configuration is changed in one time step.

The long time behaviour of a dynamical system is described by its attractors. Attractors are states of the system towards which the system is attracted. The system can converge to a particular fixed point attractor or to a periodic limit cycle attractor. If the system expresses chaotic behavior then its attractors are called strange. Limit sets have been introduced as possible formalizations of attractors in the theory of cellular automata. A limit set of a cellular automaton consists of all the configurations that can occur after arbitrarily long computations.

A cellular automaton is called nilpotent if its limit set contains just one configuration. Using a compact topology defined on the set of configurations one can show that, if a CA is nilpotent, then there is an upper bound n such that every configuration turns into the only configuration of the limit set in at most n time steps (see [2] for the proof of this fact). The nilpotency can be characterized also using the graphs of cellular automata. The graph of a cellular automaton is an infinite digraph, whose nodes are the configurations of the automaton, and whose arcs express the transitions between the configurations. Obviously the graph of a nilpotent cellular automaton is connected. Also the converse is true: if there is just one component in the graph, then the automaton is nilpotent. In [11] it was proved that, if the graph has more than one component, then the number of components is uncountable.

* Received by the editors October 22, 1990; accepted for publication (in revised form) June 13, 1991.
† Mathematics Department, University of Turku, 20500 Turku, Finland.

The nilpotency problem consists of deciding whether a given CA is nilpotent or not. It was shown in [2] that the nilpotency problem is undecidable for two- and higher dimensional CA. The purpose of the present work is to prove the same result for one-dimensional CA. In [2] a known undecidable problem, the so-called tiling problem, was reduced to the nilpotency problem. This method does not work in one-dimensional case since the one-dimensional tiling problem is decidable. We can, however, use some two-dimensional tilings also in connection with one-dimensional CA. One can namely consider the space-time diagram of a one-dimensional CA as a tiling of the plane. In this case the tiling is locally deterministic in one dimension. It is easy to reduce the tiling problem of locally deterministic tile sets to the nilpotency problem. The difficult part is to show undecidable the tiling problem of locally deterministic tiles. To prove this, we use a modification of Robinson's proof of the undecidability of the general tiling problem [8].

2. Basic definitions. Formally, a *cellular automaton* (CA) is a quadruple $\mathcal{A} = (d, S, N, f)$, where d is a positive integer indicating the *dimension* of $\mathcal{A}$, S is a finite *state set*, N is a *neighborhood vector*

$$N = (\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n)$$

of n different elements of $\mathbb{Z}^d$ and f is the *local rule* of the CA presented as a function from S^n into S. The *neighbors* of a cell situated in $\bar{x} \in \mathbb{Z}^d$ are the cells in positions

$$\bar{x} + \bar{x}_i \quad \text{for } i = 1, 2, \cdots, n.$$

A *configuration* of a CA $\mathcal{A} = (d, S, N, f)$ is a function

$$c : \mathbb{Z}^d \rightarrow S$$

that assigns states to all cells. Let $\mathcal{C}$ denote the set of all configurations. The local rule f determines the global function

$$G_f : \mathcal{C} \rightarrow \mathcal{C}.$$

At each time step a configuration c is transformed into a new configuration $G_f(c)$ where

$$G_f(c)(\bar{x}) = f(c(\bar{x} + \bar{x}_1), c(\bar{x} + \bar{x}_2), \cdots, c(\bar{x} + \bar{x}_n))$$

for all $\bar{x}$ in $\mathbb{Z}^d$. For any configuration c the sequence $c, G_f(c), G_f^2(c), \cdots$ is called the *orbit* of c. For each state s in S let *conf*(s) denote the homogeneous configuration where all the cells are in the same state s.

The *limit set* $\Lambda[\mathcal{A}]$ of a CA $\mathcal{A} = (d, S, N, f)$ contains all the configurations that can occur after arbitrarily many computation steps. Define

$$\begin{aligned} \Lambda^{(0)} &= \mathcal{C}, \text{ and} \\ \Lambda^{(i)} &= G_f(\Lambda^{(i-1)}) \text{ for } i \geq 1. \end{aligned}$$

Then the limit set of $\mathcal{A}$ is

$$\Lambda[\mathcal{A}] = \bigcap_{i=0}^{\infty} \Lambda^{(i)}.$$

Every configuration $c \in \Lambda[\mathcal{A}]$ has a predecessor c' (that is, a configuration c' such that $G_f(c') = c$) that is also in $\Lambda[\mathcal{A}]$. This can be proved easily using the compact topology of $\mathcal{C}$ (see [2] for the definition of the topology). This means that for every configuration c in the limit set there is a countably infinite sequence of configurations $c_0, c_1, \cdots$ such that $c = c_0$ and $G_f(c_{i+1}) = c_i$ for every $i \geq 0$.

It is easy to see that the limit set can never be empty. Indeed, every homogeneous configuration $conf(s)$ remains homogeneous during the operation of the automaton. Because the state set is finite, there are only finitely many homogeneous configurations. This means that some $conf(s)$ must turn back into $conf(s)$ in k time steps, for some $k \geq 1$. Then obviously $conf(s)$ is in the limit set.

If the limit set of a CA contains just one configuration then the CA is called *nilpotent*. It was shown in [2] that if a CA is nilpotent then there exists an integer n such that every configuration turns into the unique configuration of the limit set in at most n time steps.

In [11] Podkolzin studied *graphs* of cellular automata. The graph of a CA $\mathcal{A}$ is an infinite, directed graph. Its nodes are the configurations of $\mathcal{A}$, and for every $c \in \mathcal{C}$ there is an arc from c to $G_f(c)$.

Let us consider the components of the graph. If the CA $\mathcal{A}$ is nilpotent then its graph has only one component, since every configuration is connected to the unique configuration of the limit set. Conversely, suppose that the graph of $\mathcal{A}$ contains just one component. Let $conf(s)$ be a homogeneous configuration that is contained in the orbits of all configurations. Such a configuration must exist because there is only one component in the graph. In [2] it was proved that, if $\Lambda[\mathcal{A}] \neq \{conf(s)\}$, then there exists a configuration c whose orbit does not contain $conf(s)$, which contradicts the selection of $conf(s)$. We conclude that $\Lambda[\mathcal{A}] = \{conf(s)\}$, and the CA $\mathcal{A}$ is nilpotent.

It was shown above that a CA $\mathcal{A}$ is nilpotent if and only if its graph has only one component. In [11] it was proved that the graph of any CA contains either one component or uncountably many components. Consequently, the graph of every non-nilpotent automaton has an uncountable number of components.

It is a natural question to ask what kind of local rule makes a cellular automaton nilpotent. The problem of testing whether a given local rule defines a nilpotent CA is called the *nilpotency problem*. In [2] it was shown that the nilpotency problem is undecidable for d-dimensional CA where $d \geq 2$. In the following the same result is proved for one-dimensional CA. The undecidability of the nilpotency problem plays a central role in [6] in the proof of the Rice's theorem for CA limit sets. It is described in [6] how the nilpotency problem can be reduced to the problem of testing any nontrivial property of limit sets. Consequently, every nontrivial property of limit sets is undecidable.

Tilings of the plane play an important role in our proof. Suppose we are given a finite set of unit squares with colored edges, the tiles. The tiles are placed with their edges horizontal and vertical. We have infinitely many copies of all the tiles and we want to tile the entire plane using the copies, without rotating any of them. In a valid tiling the abutting edges of adjacent tiles must have the same color. The *tiling problem* consists of deciding whether the plane can be tiled with a given collection of tiles. The tiling problem was proved undecidable by Berger [1]. A simpler proof was given later by Robinson [8]. In [2] the tiling problem was applied to prove the undecidability of the nilpotency problem for two- and higher dimensional CA.

In one-dimensional case certain restricted types of tile sets will be used. We call a tile set *NW-deterministic* if for any pair (A, B) of tiles there is at most one tile C such that the lower edge of A has the same color as the upper edge of C, and the right edge of B has the same color as the left edge of C. This means that the tiles A, B and C can form

the pattern of Fig. 1 without violating the tiling rule. The pair NW of letters refers to the directions north and west. In a NW-deterministic tile set the northern and western neighbors of a tile define the tile uniquely, provided there exists a matching tile at all.

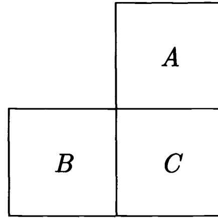


FIG. 1. *For every A and B there is at most one matching C.*

For every NW-deterministic tile set $\mathcal{T}$ a partial function $\Psi_{\mathcal{T}} : \mathcal{T}^2 \rightarrow \mathcal{T}$ can be defined by

$$\Psi_{\mathcal{T}}(A, B) = \begin{cases} C & \text{if } A, B \text{ and } C \text{ match each other in Fig. 1,} \\ \text{not defined} & \text{if there is no matching } C \in \mathcal{T}. \end{cases}$$

This means that the tiles A, B and C in Fig. 1 match if and only if $C = \Psi_{\mathcal{T}}(A, B)$.

The colors of the tiles are not needed any longer if the partial function $\Psi_{\mathcal{T}}$ is known. So we do not need to restrict ourselves to tiles with colored edges. Any finite set $\mathcal{T}$ with a partial function $\Psi_{\mathcal{T}} : \mathcal{T}^2 \rightarrow \mathcal{T}$ will be called a NW-deterministic tile set. A tiling of the plane with the elements of $\mathcal{T}$ is valid iff $\Psi_{\mathcal{T}}(A, B) = C$ whenever A, B and C form the pattern of Fig. 1.

In §3 we show how the tiling problem with the NW-deterministic tile sets can be reduced to the nilpotency problem of one-dimensional CA. In the remaining two §§4 and 5 we complete the proof by showing that the tiling problem remains undecidable even if we restrict ourselves to NW-deterministic tile sets. This proof looks very similar to the undecidability proof of Robinson in [8]. We only need to make sure that the tiles constructed during the proof are always NW-deterministic.

3. The nilpotency problem. The following proposition is the basis of our proof for the undecidability of the nilpotency problem.

PROPOSITION 3.1. *The tiling problem with NW-deterministic tile sets is undecidable.* The proposition will be proved in §§4 and 5.

Let us now show how the tiling problem with NW-deterministic tiles can be reduced to the nilpotency problem. Let $\mathcal{T}$ be any NW-deterministic tile set with the partial function $\Psi_{\mathcal{T}} : \mathcal{T}^2 \rightarrow \mathcal{T}$ defining valid tilings. Let q be a symbol not in $\mathcal{T}$. We construct a one-dimensional CA $\mathcal{A}_{\mathcal{T}} = \left(1, \mathcal{T} \cup \{q\}, (0,1), f_{\mathcal{T}}\right)$ whose state set is $\mathcal{T} \cup \{q\}$ and local rule $f_{\mathcal{T}}$ is defined as follows:

$$f_{\mathcal{T}}(A, B) = \begin{cases} \Psi_{\mathcal{T}}(A, B) & \text{if } A, B \in \mathcal{T} \text{ and } \Psi_{\mathcal{T}}(A, B) \text{ is defined,} \\ q & \text{otherwise.} \end{cases}$$

It is easy to see that $\mathcal{A}_{\mathcal{T}}$ is not nilpotent if and only if the tile set $\mathcal{T}$ can be used to tile the plane. Namely, suppose that the plane can be tiled legally using the tiles of $\mathcal{T}$. Let us index the positions of the tiles on the plane using integer coordinates. For every $x, y \in \mathbb{Z}$ let $T(x, y)$ denote the tile in the position (x, y). For each $t \in \mathbb{Z}$ a configuration

c_t of $\mathcal{A}_{\mathcal{T}}$ is constructed by taking the tiles in the infinite diagonal row that runs through the tile in the position $(t, 0)$ (see Fig. 2). For all integers t and i,

$$c_t(i) = T(t + i, i).$$

Because of the way the local rule $f_{\mathcal{T}}$ was defined, $c_{t+1} = G_{f_{\mathcal{T}}}(c_t)$ for each $t \in \mathbb{Z}$. This means that the configurations c_t are in the limit set of $\mathcal{A}_{\mathcal{T}}$. The limit set contains also *conf* (q), so that $\mathcal{A}_{\mathcal{T}}$ is not nilpotent.

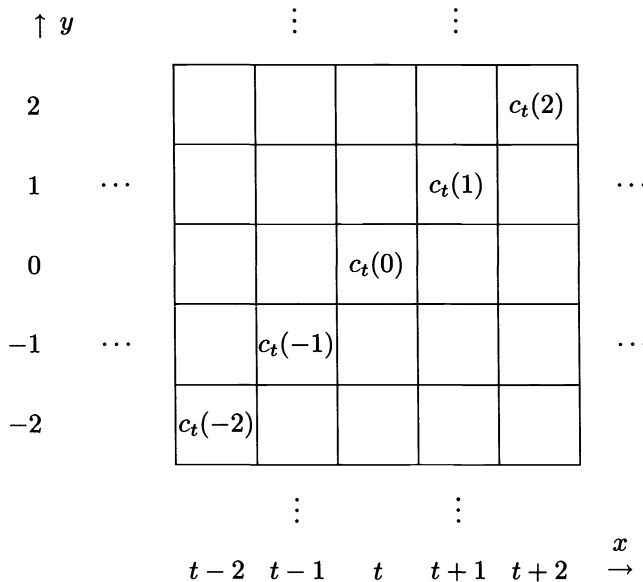


FIG. 2. *The tiles defining the configuration c_t.*

Conversely, suppose that $\mathcal{A}_{\mathcal{T}}$ is not nilpotent. Then its limit set contains a configuration c different from *conf* (q). Without loss of generality we may suppose that $c(0) \neq q$. Because $c \in \Lambda[\mathcal{A}_{\mathcal{T}}]$ there is an infinite sequence of configurations $\cdots, c_{-2}, c_{-1}, c_0$, such that $c_0 = c$ and $G_{f_{\mathcal{T}}}(c_{t-1}) = c_t$ for every $t \leq 0$. Let us define a tiling of the upper left quadrant of the plane by placing the tile $c_t(i)$ to the position $(t + i, i)$ for every $t \leq 0$ and $0 \leq i \leq -t$. The tiling is valid. This follows from the way the local rule of $\mathcal{A}_{\mathcal{T}}$ was defined. Because we can tile one quadrant of the plane, we can tile the whole plane.

If there were an algorithm for deciding whether a given one-dimensional CA is nilpotent, then this algorithm applied to automata $\mathcal{A}_{\mathcal{T}}$ would solve the tiling problem of NW-deterministic tile sets. From Proposition 3.1 we get Theorem 3.2.

THEOREM 3.2. *The nilpotency problem of one-dimensional CA is undecidable.* □

Note that in the proof of Theorem 3.2 only CA with the neighborhood vector $(0, 1)$ containing just two elements are used. Since a two-element neighborhood is contained in every nontrivial neighborhood (that is, a neighborhood containing more than one element), we conclude that the nilpotency problem remains undecidable even if we restrict ourselves to the class of CA using the neighborhood vector N, for every fixed nontrivial neighborhood vector N.

In [6] a more general result is proved, showing that every nontrivial property of limit sets is undecidable. In its proof the fact that the CA $\mathcal{A}_{\mathcal{T}}$ contains a spreading state q is needed. (A state s is called spreading if every cell whose neighborhood contains a cell in state s is turned into the state s on the next time step.)

In the subsequent sections we prove Proposition 3.1 that was needed in the proof of Theorem 3.2.

4. The basic tiles. Our proof of Proposition 3.1 follows the lines of the proof of the same result for general tile sets in [8]. We only need to take care that the tile set constructed in the proof is NW-deterministic.

First we note that we may weaken the restrictions made for NW-deterministic tile sets and allow the tile A' in Fig. 3 to affect the choice for the tile C. So we have a partial function $\Psi_\mathcal{T}$ of three arguments instead of two. In a valid tiling every four tiles forming the pattern of Fig. 3 must satisfy $\Psi_\mathcal{T}(A', A, B) = C$. This modification is done in order to make our construction more readable.

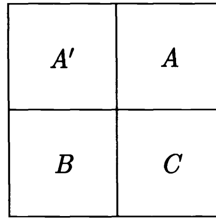


FIG. 3. *For every A, A', and B there is at most one matching C.*

If the tiling problem for NW-deterministic tile sets is undecidable after the modification, then it is also undecidable when the original definition is used. For every tile set $\mathcal{T}$ with a three argument partial function $\Psi_\mathcal{T}$ we construct a new tile set $\mathcal{T}' = \mathcal{T}^2$ with the two argument function $\Psi_{\mathcal{T}'}$ given by

$$\Psi_{\mathcal{T}'}\big((A, A'), (B, B')\big) = \big(\Psi_\mathcal{T}(A', A, B), B\big).$$

The second component of a pair $(A, A') \in \mathcal{T}'$ is used to remember the left neighbor of the tile. It is not difficult to see that there is a valid tiling with the tiles of $\mathcal{T}$ if and only if there is a tiling with the tiles of $\mathcal{T}'$.

First a NW-deterministic set of basic tiles will be described. The tiles have the property that every valid tiling of the plane is nonperiodic, that is, there exist no translations of the plane that leave the tiling unchanged. The basic tiles are the same that were used in [8], and they resemble also the tiles used in [5]. Only a small change is needed to make the tiles NW-deterministic. The tile set is by no means minimal—it is possible to construct a NW-deterministic, nonperiodic set containing only 16 tiles [4]. However, the basic tile set described below is better suited for our purpose.

The tiles contain arrows. The arrows are first horizontal and vertical. Later on also diagonal arrows will be introduced. On a valid tiling each arrow head must meet an arrow tail on the neighboring tile. The seven basic tiles are represented in Fig. 4. The tiles may be rotated, so that the total number of different tiles is 28. Each tile contains central arrows in the middle of their sides and possibly some additional side arrows.

The first tile containing arrow heads on all four sides is called a cross. The cross is said to face to the two directions where it has the side arrows. The cross drawn in Fig. 4 faces up and to the right.

The other tiles are called arms. Every arm contains a principal arrow running across the tile from one side to the opposide side. The arm is said to point to the direction of its principal arrow. The arm may have also a side arrow parallel to the principal arrow. The side arrow may be on either side of the principal arrow. Each arm has also two meeting

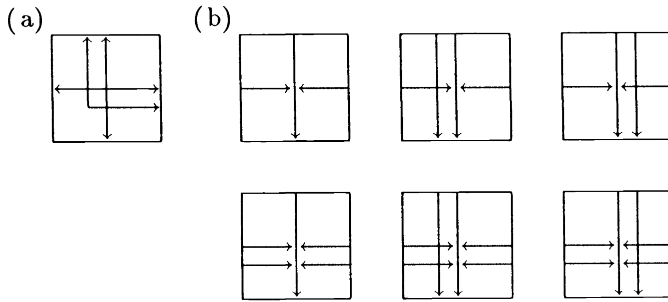


FIG. 4. *The seven types of basic tiles:* (a) *a cross and* (b) *arms.*

arrow heads at right angles to the principal arrow. If the two meeting arrow heads have side arrows then they must be toward the head of the principal arrow.

As in [8], a cross must be forced to occur in alternate columns in alternate rows. This is accomplished by adding a new component—a parity tile—to every basic tile. The four parity tiles are depicted in Fig. 5. In every valid tiling of the plane the parity tiles alternate both horizontally and vertically. If the coordinates of the columns and rows are shifted suitably the lower left parity tile occurs in the intersections of even numbered rows and even numbered columns.

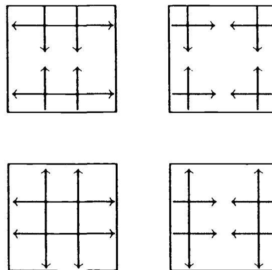


FIG. 5. *The parity tiles.*

The parity tiles are attached to the basic tiles so that the parity tile at the lower left in Fig. 5 is attached only to the crosses. The parity tile at the lower right is attached to vertical arms and the upper left tile is attached to horizontal arms. The parity tile at the upper right can be attached to any of the basic tiles. Every basic tile has two possibilities for the parity tiles so that the total number of tiles becomes 56. The use of parity tiles forces the tiles in the intersections of even numbered rows and even numbered columns to be crosses.

Let us now study the possible tilings with the set of 56 tiles described above. The set is exactly the same that was used by Robinson in [8], so that his analysis of possible tilings can be directly used.

For each positive integer n, four $(2^n - 1)$-squares are defined recursively. A cross with the parity tile at the lower left in Fig. 5 is a 1-square. There are four 1-squares because there are four possible orientations of the cross.

For every $n \geq 2$ a $(2^n - 1)$-square consists of four $(2^{n-1} - 1)$-squares separated by a cross and rows of arms leading radiately out from the center (see Fig. 6). The cross in the center is called the central cross of the $(2^n - 1)$-square. There are four $(2^n - 1)$-squares because the orientation of the central cross is arbitrary. The $(2^n - 1)$-square is said to

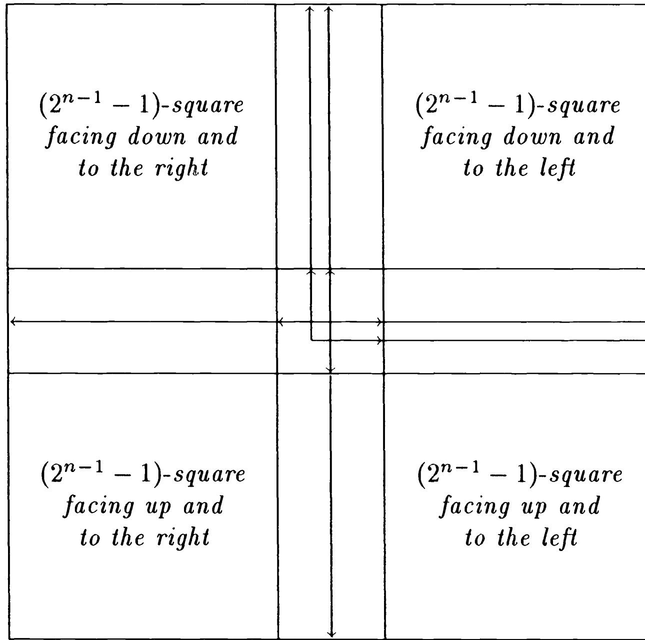


FIG. 6. *Constructing a* $(2^n - 1)$*-square.*

face to the same direction as its central cross. An example of a 7-square is depicted in Fig. 7. It is easy to see that the tiling property is satisfied inside the $(2^n - 1)$-squares.

The following lemma was proved in [8].

LEMMA 4.1. *In any valid tiling of the plane using the tiles above there must be a* $(2^n - 1)$*-square on the plane for every* $n \geq 1$.

According to Lemma 4.1 every legal tiling of the plane is nonperiodic. This follows from the fact that a $(2^n - 1)$-square is not invariant under any horizontal or vertical translation of length less than $2^n - 1$.

To make the tile set NW-deterministic, diagonal arrows running from the upper left corner to the lower right corner of the tiles must be added. Each tile contains exactly one diagonal arrow, and the arrow is labeled either *Ver* or *Hor*. The head and the tail of the arrow may have different labels. In a valid tiling the tail of the diagonal arrow on each tile must have the same label as the head of the diagonal arrow on its upper left cornerwise neighbor.

On each horizontal arm the diagonal arrow has label *Hor* on its tail and *Ver* on its head (see Fig. 8). On vertical arms the labels are the opposite: *Ver* on the tail and *Hor* on the head. The diagonal arrows of the crosses have always the same label on their tail and head. For each cross there are two possibilities to choose the diagonal arrow. It may be labeled either *Hor* or *Ver*. The total number of tiles is 64 after the diagonal arrows are added.

The diagonal arrows force the horizontal and vertical arms to alternate on each diagonal row of tiles that is running down and to the right. There may be any number of crosses between the arms, but the next arm after a horizontal arm must always be vertical and vice versa. It is not difficult to prove that this is always the case in the $(2^n - 1)$-squares defined above. So the following lemma holds true.

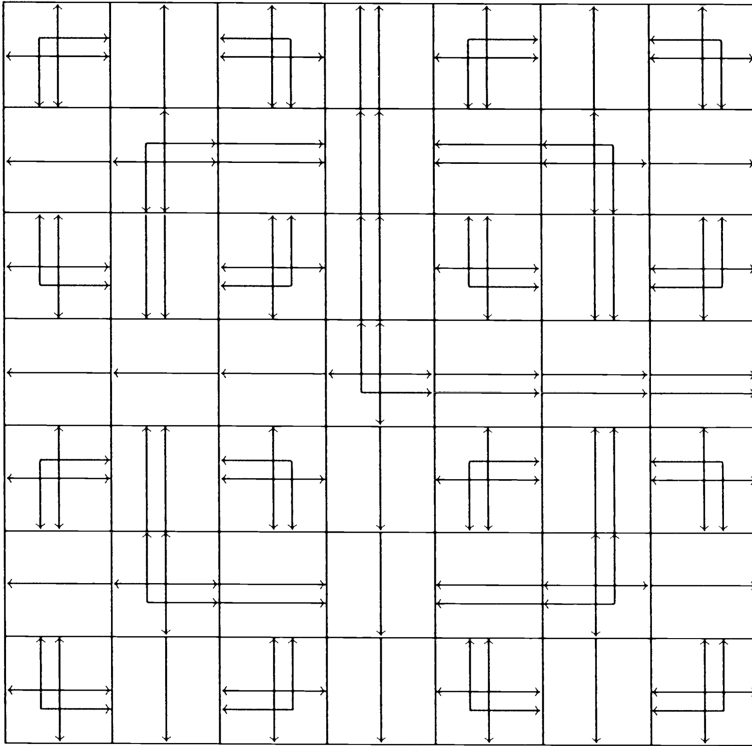


FIG. 7. *The 7-square facing up and to the right. Only the principal arrows and their side arrows of the arms are drawn.*

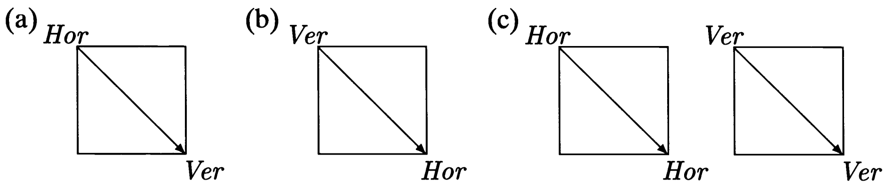


FIG. 8. *The diagonal arrows on* (a) *horizontal arms,* (b) *vertical arms, and* (c) *crosses.*

LEMMA 4.2. *For every $n \geq 1$ the tiling is valid on the $(2^n - 1)$-squares even after the diagonal arrows are added to the tiles.*

It follows from Lemma 4.2 that the tile set can be used to tile the plane. Naturally, the diagonal arrows do not affect Lemma 4.1, so that all such tilings are nonperiodic.

The diagonal arrows are needed to make the tile set NW-deterministic.

LEMMA 4.3. *The tile set constructed above is NW-deterministic.*

Proof. First note that the parity tiles of Fig. 5 are NW-deterministic (actually every tile defines both its right and its lower neighbors uniquely, so that it is enough to look at either A or B of Fig. 3 to determine C). Let us then consider the basic tiles of Fig. 4 with the diagonal arrows of Fig. 8. We have different cases depending on what kind of tiles the tiles A and B of Fig. 3 are.

Suppose first that the arrows on the lower side of A and the right side of B are both pointing away from C. Then the tile C must necessarily be a cross. The orientation of the cross is uniquely determined by the side arrows of A and B. The label on the diagonal arrow of C is fixed by the diagonal arrow of A'.

Suppose then that the lower side of A has an arrow pointing away from C, while the arrow on the right side of B is pointing towards C. Then C must be an arm pointing upwards. Again the side arrows of C are uniquely determined by the side arrows of A and B. The case when A has an arrow pointing downwards and B an arrow pointing to the left is symmetric. Then C must be an arm pointing to left.

Finally, suppose that both A and B have arrows pointing towards C. In this case the diagonal arrow of A' is needed. If the head of this arrow is labeled *Hor* then C must be an arm pointing to the right, if the label is *Ver* then C is pointing down. The side arrows are again fixed by the side arrows of A and B. □

5. The undecidability of the tiling problem of NW-deterministic tiles. In this section the halting problem of Turing machines started on a blank tape is reduced to the tiling problem of NW-deterministic tile sets. Turing machines will be simulated on tilings with the basic tile set of §4. The idea is very similar to the one used by Robinson in [8].

A Turing machine consists of a finite state set S and a finite alphabet A of tape symbols. The tape is infinite in both directions. A special tape symbol $a_0 \in A$ is called *blank*. There are two special states in S: s_0 is the *initial state* and s_h is the *halting state* of the machine. The machine works under some rules of the form

$$(a, s) \rightarrow (a', s', d), \text{where } a, a' \in A, s, s' \in S \text{ and } d \in \{L, R, S\}.$$

The rule says that, if the machine is in state s and its read-write head is scanning the tape symbol a, then it will overprint a by a', change its state into s' and move its read-write head as d indicates. If d is S then the head remains in the current position; if d is L then the head moves one symbol to the left; and if d is R then the head moves one symbol to the right. The Turing machines considered here are deterministic. This means that for every pair (a, s) there is exactly one rule of the form above, except if $s = s_h$, in which case there is no rule at all.

Initially the Turing machine is in the initial state s_0, and its read-write head is in the position 0 of the tape. The tape contains only blank symbols a_0. The halting problem asks whether the Turing machine eventually is changed into the halting state. The halting problem is known to be undecidable—there is no algorithm that could decide of a given machine whether it halts or not.

In the following Turing machine computations will be simulated on tilings of the plane. We show how to construct for any given deterministic Turing machine a NW-deterministic tile set based on the basic tiles of §4 such that there is a legal tiling if and only if the Turing machine does not halt when started on a blank tape. The simulation of the computation is straightforward. The tape of the Turing machine is represented on a diagonal row of tiles. Consecutive rows represent the content of the tape on consecutive time steps. Signals are used to pass the tape symbols from one time step to another. Also the read-write head of the Turing machine is included in one tile, and its operation can be simulated using signals. To make the tiling impossible if the Turing machine halts tiles representing the halting state s_h are omitted.

However, we have to guarantee that on each tiling of the plane there is a simulation going on, that is, there is a read-write head of the Turing machine somewhere on the plane. In fact, in our construction there will be infinitely many simulations at the same time. Among them there are arbitrarily long simulations, so that the halting state is reached in some of them, if the Turing machine halts.

To obtain this we have to add some new features to the basic tiles. First, by coloring the arrows of the basic tiles we can separate arbitrarily large hollow squares, so-called

borders, on each legal tiling. Inside every border the Turing machine is simulated. However, the borders can be situated inside each other. To prevent the computations from interfering with each other only so-called free rows and columns are used. A row or column is free inside a border if it does not intersect with any smaller border inside. In order to recognize free rows and columns in a NW-deterministic way a new set of signals, called obstruction signals, are used.

Let us go into details. Remember that every $(2^{n+1}-1)$-square contains four (2^n-1)-squares in the corners, and a cross in the center with rows of arms radiating out from it. Each of the crosses in the centers of the $(2^n - 1)$-squares is facing two of the others. These crosses together with the arms in between them form a hollow square whose side has the length of $2^n + 1$ tiles. This hollow square will be called a 2^n-*border*.

A $(2^{n+1} - 1)$-square contains one 2^n-border, four 2^{n-1}-borders, sixteen 2^{n-2}-borders, etc. The only cross inside the $(2^{n+1} - 1)$-square which does not belong to a border within this square is the one in the center. The 2^n-border does not intersect any other 2^n-border, and the only larger border it intersects is the 2^{n+1}-border one of whose corners is at the center of the 2^n-border. So two borders can intersect only if the side of one is twice the side of the other. This is true inside all $(2^n - 1)$-squares.

The basic tiles are now modified by coloring the side arrows red or green, just like in [8]. In a valid tiling two meeting side arrows must always have the same color. In each tile only one color is used horizontally and one color vertically. In a cross, the same color is used both vertically and horizontally, while in an arm which has side arrows both ways, the horizontal and vertical side arrows have different colors. The side arrows of those crosses that are attached to the parity tile at the lower left corner of Fig. 5 are always colored green. So the crosses in alternate columns in alternate rows are green.

The colors go completely around the borders, so that each border is either green or red. Two intersecting borders are always of different colors. The 2-borders are forced to be green by the constraint above for the crosses in alternate columns in alternate rows. So 4-borders are red, 8-borders green, etc. Every 2^n-border is green when n is odd, but red when n is even.

The coloring of the side arrows should be NW-deterministic. To obtain this also the arrows without a side arrow are colored green or red. The rule of this coloring is very simple: In each tile all horizontal (principal as well as side) arrows have the same color. Similarly, all vertical arrows have the same color. In a valid tiling the meeting arrows must have the same color, so that the colors run unchanged through the horizontal and vertical rows of the plane. The coloring is obviously NW-deterministic—the color of the horizontal arrows is determined by the left neighbor, and the color of the vertical arrows is forced by the upper neighbor. The coloring of principal arrows does not violate the restrictions given above for the colors of the side arrows, because inside every $(2^n - 1)$-square the borders of different sizes have their horizontal and vertical edges on different horizontal and vertical rows, respectively. So the coloring can be done without violating the tiling property.

Let us now forget the green borders and consider the red borders. They are exactly the 4^n-borders, for all positive integers n. Two red borders cannot intersect, but a smaller red border may lie completely within a larger one. The region within a red border but outside all red borders within it will be called a *board*. On a board we want to locate the rows and columns which run completely across the board, from outer border to outer border, without running into any of the smaller boards inside. These rows and columns will be called *free*.

Next we count the number of free rows and columns. Let F_n denote their number

in a board with the side of $4^n - 1$ tiles. Note that the positions of the 4^k-borders repeat both horizontally and vertically with the period $2 \cdot 4^k$. So the pattern of free columns of a board of side $4^n - 1$ tiles is exactly repeated in the middle of a board of side $4^{n+1} - 1$. Also halves of this pattern are repeated at the sides of the larger board, excluding the center column of the pattern. Thus $F_{n+1} = 2F_n - 1$. Because $F_1 = 3$, we get $F_n = 2^n + 1$. The number is naturally the same for the rows as well.

In order to locate the free rows and columns in a NW-deterministic way, new signals running along the rows and columns are needed. The signals will be called the *obstruction signals*, since they tell whether there is any obstruction on the line. There are two obstruction signals: one horizontal that travels to the right, and one vertical that travels down. The signals are emitted and absorbed by the red borders.

Let us consider the vertical signal (the horizontal one is symmetric). The signals on different tiles are depicted in Fig. 9. Every tile on the lower edge of a red border emits a signal downwards. These are the tiles having a red horizontal side arrow below the central level. They are the only tiles that emit signals. They can also absorb a signal coming from the tile above them. (In this case the tile both absorbs and emits a signal, so that it has the same effect as if the signal had only been transmitted through the tile.) The tiles on the upper edge of a red border, on the other hand, can only absorb signals. All other tiles transmit a signal coming from the tile above to the tile below.

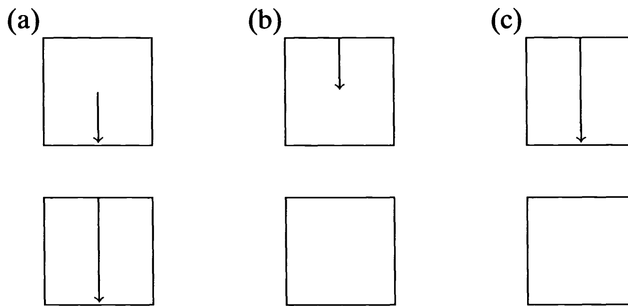


FIG. 9. *The possible vertical obstruction signals of* (a) *the tiles on the lower edge of a red border,* (b) *the tiles on the upper edge of a red border, and* (c) *all the other tiles.*

For each tile there are two possibilities for its vertical obstruction signals as depicted in Fig. 9. The tiles on the lower edge of a red border are the only tiles that necessarily contain some signal. Symmetrically, each tile also has two possibilities for the horizontal signals. The signals are NW-deterministic: The vertical signals of a tile are always uniquely determined by the signal it receives from the tile above it, and similarly its horizontal signals are forced by its left neighbor.

Let us see how the obstruction signals are used to locate the free rows and columns. Consider a 4^{n+1}-border on the plane. It contains four 4^n-borders inside. Take the one that is situated in the upper left corner. We claim that there are vertical obstruction signals on the upper edge of this 4^n-border in exactly those tiles that do not start a free column inside the 4^n-border, and that are not corners of the border.

If there is a vertical obstruction signal in a tile belonging to the upper edge of the 4^n-border, then there has to be a 4^k-border, for some $k < n$, emitting the signal somewhere between the upper edges of the 4^n- and 4^{n+1}-borders. Because the 4^k-borders repeat with the period $2 \cdot 4^k$, this means that there is a 4^k-border in the same column inside the 4^n-border. So the column is not free.

Conversely, consider a tile on the upper edge of the 4^n-border that does not start

a free column. Then it is either in the corner of the border, or in its column there is a 4^k-border, for some $k < n$, inside the 4^n-border. But the 4^k-borders repeat with the period $2 \cdot 4^k$, and the distance between the upper edges of the 4^n- and 4^{n+1}-borders is $2 \cdot 4^{n-1}$, so that there is a 4^k-border between them in the same column. The lower edge of this border emits an obstruction signal that reaches the upper edge of the 4^n-border. (On the way there can be other borders whose upper edges absorb the signal, but their lower edges always emit a new one.)

In a similar way, the horizontal obstruction signals locate the free rows inside the same 4^n-border. The free rows and columns of the other three 4^n-borders inside the bigger border will not be found using the obstruction signals. For example, the border situated in the lower left corner receives obstruction signals to every tile on its upper edge from the 4^n-border above it, so that none of the columns is recognized to be free. But this does not matter, since it is enough to locate the free columns and rows inside one 4^n-border for every n.

The 4^n-border and the free rows and columns inside form a grid (see Fig. 10) in which the operation of a Turing machine is simulated. The simulation is done in a NW-deterministic way. Let us index the rows and columns of the grid by natural numbers. The leftmost column gets the number 0, and the numbers increase to the right. The numbers of the rows increase downwards, and the number of the uppermost row is 0. Let $N = F_n + 1$ be the maximum row and column index in the grid.

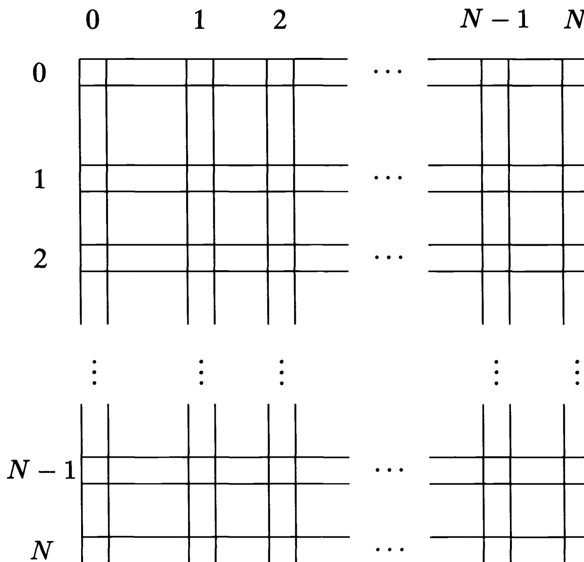


FIG. 10. *The grid where the computation of the Turing machine is simulated.*

Let us now see how a given deterministic Turing machine can be simulated in the grid defined by the 4^n-border in a NW-deterministic way. For any natural numbers k and m, $0 \leq k, m \leq N$, let $T(k, m)$ denote the tile in the intersection of the kth row and mth column of the grid. The tiles $T(k, m)$ will be called the *intersection tiles*. Each intersection tile can contain one tape symbol a from the tape alphabet of the Turing machine, and possibly a state s from the state set of the machine. The state indicates that the Turing machine is in state s, and its read-write head is scanning the tape symbol contained in the tile. The intersection tiles can send signals down and to the right. The

signals are transmitted unchanged to the next intersection tile by the tiles in the corridors between them. The corridors do nothing but transmit the signals.

The simulation of the Turing machine is started at the tile $T(0,0)$. It contains the blank tape symbol and the initial state of the Turing machine, expressing the original situation where the read-write head is scanning the blank symbol in the initial state. The subsequent configurations of the Turing machine are represented on the diagonal rows of intersection tiles. For $k > 0$ the tiles

$$T(2k,0) \quad T(2k-1,1) \quad T(2k-2,2) \quad \cdots \quad T(1,2k-1) \quad T(0,2k)$$

(or those $T(x, 2k-x)$ of them where $x, 2k-x \leq N$) represent the positions $-k, -k+1, \cdots, k-1, k$ of the tape of the Turing machine after k time steps. They contain the tape symbols in the corresponding positions of the tape. Also the state of the machine, representing the read-write head, is contained in the tile that corresponds to the tape position the machine is scanning after k time steps. Note that only every other diagonal row represents the Turing machine configurations.

This representation of the configurations is accomplished as follows. Let a be the tape symbol that is contained in a tile $T(x, 2k-x)$ of the grid, where $0 \leq x < N$. The tile sends a signal a_D downwards. The subscript D denotes the fact that the symbol is traveling down. When the signal comes to the next intersection tile $T(x+1, 2k-x)$, it is changed into a_R and sent to the right. When a_R reaches the tile $T(x+1, 2k+1-x)$ (which represents the same tape position as $T(x, 2k-x)$, only one time step later) the tile knows that its tape symbol is a. This works if the tile $T(x, 2k-x)$ does not contain the read-write head of the Turing machine. If some state s is at the tile, representing the read-write head, then the signal a'_D is sent instead, where a' is the symbol that the Turing machine overprints a with. This is how the tiles $T(x, 2k-x)$ with $x, 2k-x > 0$ get their tape symbols. The tiles $T(0, 2k)$ and $T(2k, 0)$ always contain the blank tape symbol a_0.

Let us now consider the read-write head and state s of the Turing machine in some tile $T(x, 2k-x)$. Let a be the tape symbol in the same tile, and let s' be the state the Turing machine is changed into when scanning a in state s. Let $d \in \{L, R, S\}$ denote the direction where the head moves. Suppose s' is not the halting state of the Turing machine.

Suppose first that $d = L$. In this case a signal $(s'_L, 1)$ is sent downwards from the tile $T(x, 2k-x)$, provided $x < N$. The subscript of s' denotes d, and the number 1 expresses the fact that the signal is on the first part of its way to the next tape position (every signal goes through two corridors between the intersection tiles before reaching its goal). When the signal comes to the next intersection tile $T(x+1, 2k-x)$ it is changed into $(s'_L, 2)$ and sent again down (if $x+1 < N$). The next intersection tile it meets is $T(x+2, 2k-x)$ that represents the Turing machine tape position situated one step to the left from the position of $T(x, 2k-x)$, at the next time instant. So this tile knows it contains the read-write head of the machine, which is currently in state s'.

Similarly, if $d = R$ then the signal $(s'_R, 1)$ is sent to the right to the tile $T(x, 2k+1-x)$. This tile sends $(s'_R, 2)$ again to the right. So the tile $T(x, 2k+2-x)$ gets the state s' (this happens naturally only if $2k+2-x \leq N$), which is correct since it represents the tape position one step to the right from the position represented by $T(x, 2k-x)$.

Finally, if $d = S$ then a signal $(s'_S, 1)$ is sent down from $T(x, 2k-x)$, and a signal $(s'_S, 2)$ to the right from the tile $T(x+1, 2k-x)$. So the tile $T(x+1, 2k+1-x)$ receives, quite correctly, the state s'.

It is obvious that the signals described above make sure that the diagonal rows represent the configurations of the Turing machine correctly, if the Turing machine does not

halt. The signals are bounded inside the grid, because neither the tiles on the lower edge ever send signals downwards, nor do the tiles on the right side send any signals to the right. If the read-write head of the Turing machine tries to get out of the grid (this can happen after $\lfloor N/2 \rfloor$ time steps from the beginning), then the head simply disappears, and in the subsequent diagonal rows only the tape symbols are contained.

The signals are NW-deterministic—they travel only down and to the right. Because the Turing machine simulated is deterministic, also the signal a'_D, which is created from the tape symbol a in the tile that contains the read-write head as described above, is uniquely determined.

The halting state s_h of the Turing machine is never contained in any tile. Also there are no signals carrying the halting state. If at some tile containing the read-write head the machine is turned into the state s_h, then no signal (s_{h_d}, i) can be sent, and the tiling becomes impossible.

Suppose the given Turing machine halts in K time steps after it has been started. Choose $n \geq \log_2(2K)$. Consider the grid composed of a 4^n-border and the free rows and columns inside. According to Lemma 4.1 such a grid must occur in every valid tiling. The maximum index of the rows and columns in the grid is

$$N = F_n + 1 = 2^n + 2 \geq 2K + 2.$$

The read-write head of the Turing machine cannot go out of the grid during the first $\lfloor N/2 \rfloor > K$ time steps of the simulation, since the portion of the tape represented in the grid keeps expanding to both directions. This means that the halting state s_h is encountered in the grid. But no tile can contain the halting state, so that the tiling is impossible.

On the other hand, if the Turing machine never halts, then the simulation is possible in the grids inside the 4^n-borders, for every n. This means that the plane can be tiled with the tiles.

We have proved that the given Turing machine halts if and only if the NW-deterministic tile set constructed in §§4 and 5 can be used to form a valid tiling. Since the halting problem of deterministic Turing machines started on the blank tape is undecidable, this proves Proposition 3.1, and completes the proof of Theorem 3.2. □

6. Concluding remarks. We have shown that the nilpotency problem of cellular automata is undecidable even in case of one-dimensional CA. The proof was based on the close relation between computations of one-dimensional CA and tilings of the plane with tile sets that are locally deterministic in one dimension. Such tilings can be considered as space-time diagrams of one-dimensional CA. We propose the use of this close relation also in solving other open problems concerning one-dimensional CA. Tilings, being static objects, are easier to handle than dynamic CA. The construction presented above could have been done without considering tilings. For example the signals running across the plane along rows and columns correspond to the traditional signals of CA transferring information to the right or left.

The specific property of tilings that corresponds to the nilpotency of CA is the existence of legal tilings. The undecidability of this problem is proved using a straightforward extension of Robinson's proof for the undecidability of the existence of legal tilings with arbitrary tile sets [8]. Robinson's construction has to be changed only to make the tile sets obtained remain locally deterministic.

Acknowledgments. This work has greatly benefited from discussions with Karel Culik II, Lyman Hurd, and Anatoli Bolotov, to all of whom I express my gratitude.

REFERENCES

[1] R. BERGER, *The undecidability of the domino problem*, Mem. Amer. Math. Soc., 66 (1966).
[2] K. CULIK II, J. PACHL, AND S. YU, *On the limit sets of cellular automata*, SIAM J. Comput., 18 (1989), pp. 831–842.
[3] L.P. HURD, *Formal language characterizations of cellular automaton limit sets*, Complex Systems, 1 (1987), pp. 69–80.
[4] L.P. HURD, J. KARI, AND K. CULIK, *The topological entropy of cellular automata is uncomputable*, to appear.
[5] J. KARI, *Reversibility and surjectivity problems of cellular automata*, to appear.
[6] ———, *Rice's theorem for limit sets of cellular automata*, to appear.
[7] E.F. MOORE, ed., *Sequential Machines. Selected Papers*, Addison-Wesley, Reading, MA, 1964.
[8] R.M. ROBINSON, *Undecidability and nonperiodicity for tilings of the plane*, Invent. Math., 12 (1971), pp. 177–209.
[9] A. SALOMAA, *Computation and Automata*, Cambridge University Press, Cambridge, 1985.
[10] S. WOLFRAM, *Theory and Applications of Cellular Automata*, World Scientific Publishing, Singapore, 1986.
[11] A.S. PODKOLZIN, *O povedenijah odnorodnijh struktor*, Probl. kibernetiki, 31 (1976), pp. 133–166, in Russian.

SIAM J. COMPUT.
Vol. 21, No. 3, pp. 587–599, June 1992

© 1992 Society for Industrial and Applied Mathematics
010

LEARNING MONOTONE BOOLEAN FUNCTIONS BY UNIFORMLY DISTRIBUTED EXAMPLES*

QIAN PING GU† AND AKIRA MARUOKA‡

Abstract. Valiant introduced a new computational model of concept learning by examples, gave the definition of learnability of classes of Boolean functions, and derived algorithms for learning specific classes of Boolean functions. Using his model as a base, the authors show that the class of Boolean functions expressed by monotone disjunctive normal form formulae with at most a fixed number of monomials and the class of Boolean threshold functions are polynomial time learnable when the examples are generated according to the uniform distribution.

Key words. computational complexity, Boolean function, concept learning, learning by examples, monotone Boolean formula

AMS(MOS) subject classifications. 68C25, 68C05, 68G05

1. Introduction. Recently, Valiant [V1], [V2] introduced a new computational model of concept learning, gave the definition of learnability of classes of Boolean functions, and derived algorithms for learning specific classes of Boolean functions. A class of Boolean functions is said to be learnable if there exists a polynomial time learning algorithm to learn any Boolean function in the class: when given some partial information about some unknown target Boolean function in the class, the polynomial time learning algorithm outputs with high likelihood a Boolean formula that is a reasonably accurate approximation to the target Boolean function. A special form of the learning model is called the distribution free model, in which information about a function f to be learned is given through examples of the form $(v, f(v)), v \in \{0,1\}^n$, generated according to some fixed but unknown probability distribution. Assuming the distribution free model, some classes of Boolean functions, or class of Boolean formulae, are proved to be learnable and, on the assumption that $RP \neq NP$, others are proved to be not learnable [V1], [KLPV1], [KLPV2], [PV]. In particular, it is shown in [KLPV1] and [PV] that the following classes are not learnable unless $RP = NP$: the class of disjunctive normal form formulae with at most k monomials, denoted **k-term-DNF**; the class of monotone disjunctive normal form formulae with at most k monomials, denoted **k-term-MDNF**; the class of monotone disjunctive normal form formulae, in which each variable occurs at most once and at most k monomials appear, denoted **k-term-μMDNF**; the class of Boolean threshold functions, denoted **TH**, and so on.

In the definition of learnability examples are assumed to be drawn on some unknown arbitrary probability distributions. This assumption, however, may be too strong in some practical situations. In the case where learning a class is thought to be intractable, or no learning algorithm exists, it is reasonable to try to find feasible learning algorithms that work for specific natural input distributions of the examples. In this paper we restrict ourselves to the case where examples are generated according to the uniform distribution and give a polynomial time learning algorithm for **k-term-MDNF** and one for **TH**, both of which, as mentioned above, are known not to be learnable in the distribution free model unless $RP = NP$. This suggests that learning under uniform distribution alone is quite limited. It is shown in [KLPV1] that **μMDNF**, the class of monotone

*Received by the editors December 21, 1988; accepted for publication (in revised form) June 4, 1991. This work was partially supported by a Kurata research grant.

†Institute of Software, Academia Sinica of China, P.O. Box 8718, Beijing 100080, China. Present address, Department of Electrical and Computer Engineering, University of Calgary, Calgary, T2N 1N4, Canada.

‡Department of Information Engineering, Faculty of Engineering, Tohoku University, Sendai, 980, Japan.

disjunctive normal form formulae in which each variable occurs at most once, is learnable in the uniform distribution setting while it is not in the distribution free setting on the assumption $RP \neq NP$. The same result also holds for **μDNF**, the class of disjunctive normal form formulae in which each variable occurs at most once. This is because we can relabel negated variables with new variables that denote their negations.

The main result of this paper shows that, if we restrict ourselves to the case where the number of monomials in disjunctive normal form formulae is at most a fixed number, the restriction of allowing only one occurrence of each variable can be removed in learning in the uniform distribution setting: for fixed k, **k-term-MDNF** is learnable in the uniform distribution setting. Using different approaches, the same result is proved in [KMP] and recently in [OM]. However, how to use uniformly distributed examples to find the target function is not mentioned explicitly in [KMP]. It is interesting to note that learning **k-term-μMDNF** is as hard as learning **k-term-MDNF**, and hence learning **k-term-DNF**, in the distribution free setting [KLPV1].

This paper is organized as follows. In §2, we illustrate Valiant's learning model and give the definition of learnability. On the assumption of uniform distribution of examples, we show in §3 that, for fixed k, **k-term-MDNF** is learnable, and show in §4 that **TH** is learnable.

2. Preliminaries. We first describe Valiant's learning model formally. In this paper, we restrict the objects to be learned to Boolean formulae. Let f be a Boolean function. In the following, we will use f to express a representation of f and the set $\{v|f(v) = 1\}$ as well as the function f, when the meaning can be understood by the context. Thus, for two Boolean functions f and g, we may write $f \subseteq g$, which means that $\{v|f(v) = 1\} \subseteq \{v|g(v) = 1\}$. A Boolean function f of n variables may sometimes be written as f^n to stress that it is a function of n variables. Let size(f) denote the fewest number of symbols needed to write function f or the function that formula f represents. In particular, the size of a monomial is the number of literals in it. The learning model defined by Valiant consists of a learning protocol and a learning algorithm. The former specifies the manner in which information about a function to be learned is obtained from outside and the latter is the procedure by which a formula to approximate the target function is deduced. In this paper we are exclusively concerned with the learning protocols by which positive and negative examples of a target function, denoted f, are supplied. More precisely, learning algorithms are assumed to call the following subroutines.

P-EXAMPLE: It returns a vector $v \in f^{-1}(1)$ according to the probability distribution D_f^+ on $f^{-1}(1)$.

N-EXAMPLE: It returns a vector $v \in f^{-1}(0)$ according to the probability distribution D_f^- on $f^{-1}(0)$.

In the above, D_f^+ and D_f^- denote the probability distributions according to which the positive and negative examples, respectively, are generated. When D_f^+ and D_f^- are some arbitrarily fixed unknown distributions, the model of learning is called the distribution free model of learning by examples, while when D_f^+ and D_f^- are some known distributions, the model is called the distribution specific model of learning by examples [KLPV1]. In this paper, we deal with the distribution specific model. In particular, D_f^+(respectively, D_f^-) is assumed to be the uniform distribution over the positive examples of f (respectively, the negative examples) unless otherwise stated. In what follows, D_f^+ and D_f^- will be simply written as D^+ and D^-, respectively, when no confusion arises. In Valiant's general learning model, other types of protocols, e.g., one that returns the value $f(v)$ for v the learning algorithm chooses, are also considered.

For Boolean functions f and g, let

$$D^+[f \triangle g] = \sum_{v \in f-g} D^+(v)$$

and

$$D^-[f \triangle g] = \sum_{v \in g-f} D^-(v),$$

where $f-g$ denotes $\{v|f(v) = 1\} - \{v|g(v) = 1\}$, and similarly for $g-f$. When f denotes a function to be learned and g denotes an output of a learning algorithm to approximate f, these quantities are thought of as the errors in the output. The following definition of the learnability for a class of Boolean formulae is due to Valiant [KLPV1], [V1].

DEFINITION 2.1 [V1]. *Let F be a class of Boolean formulae. F is learnable if and only if there exists a polynomial $p(\cdot, \cdot, \cdot)$ and a learning algorithm A that calls P-EXAMPLE and N-EXAMPLE such that for any n, $f^n \in F$, and $h > 1$ (error parameter) the algorithm A halts in time $p(n, size(f^n), h)$ and outputs a formula $g^n \in F$ that with probability at least $1 - 1/h$ satisfies*

$$D^+[f \triangle g] < 1/h$$

and

$$D^-[f \triangle g] < 1/h.$$

For a more complete discussion of the learning model and relevance of the definition of learnability see [V1].

For later reference, we define here the classes of formulae which we shall consider. Let the variables of f^n be denoted $x_1, \cdots, x_n$. Let the set of variables be denoted X. Given a target function f, let v denote the random variable that takes values in $f^{-1}(1)$ according to the uniform distribution D^+. Similarly, let u denote the random variable that takes values in $f^{-1}(0)$ according to the uniform distribution D^-. Let v_i denote the ith component of v and similarly for u_i. When no confusion arises, v and u also denote vectors in $\{0, 1\}^n$. A monomial is a conjunction of variables. Let $\mathrm{Var}(m)$ denote the set of variables that appear in monomial m. For a subset Y of X, let $I(Y)$ denote the set of indices of the variables in Y. Let $\log_2 x$, $\log_e x$ and 2^x be denoted $\log(x)$, $\ln(x)$ and $\exp(x)$, respectively, where e is the base of the natural logarithm. We define the following classes of formulae.

k-term-MDNF : the class of monotone disjunctive normal form formulae with at most k monomials.

Let Y be a subset of $\{x_1, \cdots, x_n\}$ and t be a positive integer. The threshold function $th_{Y,t}$ is defined to be

$$th_{Y,t}(x_1, \cdots, x_n) = \begin{cases} 1, & \text{if at least } t\ x_i\text{'} \in Y \text{ take value 1} \\ 0, & \text{otherwise.} \end{cases}$$

TH : the class of formulae that compute $th_{Y,t}$, where Y is a subset of $\{x_1, \cdots, x_n\}$ and $1 \leq t \leq |Y|$ ($|Y|$ denotes the cardinality of the set Y).

Before closing this section, we give a series of propositions which will be used in the following sections.

For $0 \leq p \leq 1$, positive integers r and t, let $LE(p,r,t)$ denote the probability of occurring at most t successes in r independent Bernoulli trials with probability of success p. Similarly, let $GE(p,r,t)$ denote the probability of occurring at least t successes in r independent trials with probability of success p.

PROPOSITION 2.2 [AV], [C], [ES]. *For* $0 \leq p \leq 1$, $0 \leq b \leq 1$, *and* $r \geq 0$,

$$LE(p,r,(1-b)pr) \leq \exp(-b^2pr/2)$$

and

$$GE(p,r,(1+b)pr) \leq \exp(-b^2pr/3).$$

For $0 < p \leq 1$, $0 < b \leq 1$, and $h > 1$, define $R(p,b,h) = \lceil 3\ln(h)/(pb^2) \rceil$. From Proposition 2.2, we can immediately get the following three propositions.

PROPOSITION 2.3. *For* $0 < p \leq 1$, $0 < b \leq 1$, $h > 1$, *and* $r \geq R(p,b,h)$,

$$LE(p,r,(1-b)pr) \leq 1/h.$$

PROPOSITION 2.4. *For* $0 < p \leq 1$, $0 < b \leq 1$, $h > 1$, *and* $r \geq R(p,b,h)$,

$$GE(p,r,(1-b)pr) \leq 1/h.$$

For h_1, $h_2 > 1$, and positive integer t, let $L(t,h_1,h_2) = \min\{r \mid \text{In } r$ independent Bernoulli trials with probability at least $1/h_1$ of success, the probability of occurring fewer than t successes is at most $1/h_2.\}$.

PROPOSITION 2.5 [C], [ES], [V1]. *For* t, h_1, $h_2 > 1$,

$$L(t,h_1,h_2) \leq 2h_1(t + \ln(h_2)).$$

3. Learning *k*-term-MDNF. In this section we show that, for a positive integer k, ***k*-term-MDNF** is learnable.

Monomial m is a prime implicant of f if and only if $m \subseteq f$ and for any m' with $m \subset m'$, $m' \not\subseteq f$, where $m \subset m'$ means $m \subseteq m'$ but $m \neq m'$. $m_1 + \cdots + m_j$ is a nonredundant prime implicant expression of f if each m_i, $1 \leq i \leq j$, is a prime implicant of f, $f = m_1 + \cdots + m_j$ and $m_1 + \cdots + m_{i-1} + m_{i+1} + \cdots + m_j \subset f$ for any i. It is easily seen that if f is a monotone function, then the disjunction of all prime implicants of f is the unique nonredundant prime implicant expression of f. When $g \subseteq f$, we say f includes g. For formula g, let

$$D^+[g] = \sum_{v \in g} D^+(v)$$

and

$$D^-[g] = \sum_{u \in g} D^-(u).$$

PROPOSITION 3.1 [OM]. *Let* $f \in$ ***k*-term-MDNF** *and* v *be the random variable taking values in* $f^{-1}(1)$ *according to* D^+. *Then for any* x_i *not appearing in* f,

$$\Pr[v_i = 1] = 1/2$$

and for any x_i appearing in some prime implicant m of f,

$$\Pr[v_i = 1] \geq 1/2 + D^+[m]/2^k.$$

Proof. It is obvious that $\Pr[v_i = 1] = 1/2$ for x_i not appearing in f. Let x_i be a variable that appears in some prime implicant m of f, and let T be the set of prime implicants of f that include the variable x_i. Let f' be the disjunction of the prime implicants in T and f'' be the disjunction of the remaining prime implicants of f. Then it is easy to see that

$$\begin{aligned}
\Pr[v_i = 1] &= \Pr[f'(v) = 1 \quad \text{and} \quad f''(v) = 0] + \frac{1}{2}\Pr[f''(v) = 1] \\
&= \Pr[f'(v) = 1 \quad \text{and} \quad f''(v) = 0] \\
&\qquad + \frac{1}{2}(1 - \Pr[f'(v) = 1 \quad \text{and} \quad f''(v) = 0]) \\
&= \frac{1}{2} + \frac{1}{2}\Pr[f'(v) = 1 \quad \text{and} \quad f''(v) = 0] \\
&\geq \frac{1}{2} + \frac{1}{2}\Pr[v \in m \quad \text{and} \quad f''(v) = 0].
\end{aligned} \tag{3.1}$$

Clearly, for each prime implicant m_j of f'', there exists x_{i_j} that appears in m_j but not in m. Therefore, since the number of prime implicants of f'' is at most $k-1$, there are at most $k-1$ such x_{i_j}'s. Let those x_{i_j}'s be $x_{i_1}, \ldots, x_{i_\ell}$, where $\ell \leq k-1$. $\{v | v \in \{0,1\}^n, v \in m\}$ is partitioned into $2^\ell \leq 2^{k-1}$ blocks according to the values of the i_1th, ..., the i_ℓth components of v's. Since $v \in m$ and $f''(v) = 0$ for v's belonging to the block consisting of v's with $v_{i_1} = 0, \ldots, v_{i_\ell} = 0$, we have

$$\begin{aligned}
\Pr[v \in m \quad \text{and} \quad f''(v) = 0] &\geq D^+[m]/2^\ell \\
&\geq D^+[m]/2^{k-1}.
\end{aligned}$$

Combining this with (3.1), we have

$$\Pr[v_i = 1] \geq 1/2 + D^+[m]/2^k. \qquad \square$$

Given $f \in$ **k-term-MDNF** and error parameter h, the prime implicant m of f is called predominant if $D^+[m] \geq 1/(kh)$. Let g be the disjunction of all the predominant prime implicants of f. Since the number of the nonpredominant prime implicants of f is at most $k-1$, we have $D^+[f \Delta g] < (k-1)/(kh) < 1/h$. We will use this fact in what follows.

Before describing the learning algorithm for **k-term-MDNF**, we give an outline of the algorithm. We determine in step 1 of the algorithm if x_j appears in a predominant prime implicant of f by calling P-EXAMPLE sufficiently many times. If there are only "few" such variables, then in step 2 we construct all possible monomials of those variables and delete such monomials that are not included by f by calling N-EXAMPLE appropriately many times (in fact, monomial m is deleted whenever N-EXAMPLE outputs a vector u such that $m(u) = 1$). It is easy to choose the predominant prime implicants of f from the remaining monomials and give as output g the disjunction of them, which is a desired approximation to target f.

On the other hand, if there are "many" variables that have been determined in step 1 to appear in predominant prime implicants of f, in step 3 we construct an output in

a different way. First, we partition the set of variables obtained in step 1 into blocks of "reasonable" size, make all of the monomials of "reasonable" size consisting of variables in exactly one of the blocks, and form all of the disjunctions of k such monomials. By restricting the way of constructing the monomials as mentioned above, we can make the run time polynomial. Finally, by calling P-EXAMPLE appropriately many times, we can choose as output g the disjunction of k monomials among those mentioned above so that $D^+[f\Delta g]$ is minimized.

The learning algorithm $A_{\mathbf{MDNF}}$ is given in Fig. 1. In the algorithm, error parameter $h > 1$ and the number of monomials $k \geq 1$ are given in advance. $\bar{0}$ denotes the all zero vector. $L(\cdot,\cdot,\cdot)$ and $R(\cdot,\cdot,\cdot)$ be as in §2. For $u, v \in \{0,1\}^n$, $u + v$ means $(u_1 + v_1, \cdots, u_n + v_n)$, where u_i and v_i are the ith components of the corresponding vectors.

Note that $|M| = 2^{|Y|}$ for M obtained in step 2.1,

$$|M| \leq (|Y|/t_2)\binom{2t_2}{\lceil \log(2kh) \rceil}$$

for M obtained in step 3.2, and that $|G| = |M|^k$ for G obtained in step 3.2. Since $k \geq 2$ holds in step 3, $R((1-1/k)/h, 1/(2k), 4|G|h)$ in step 3.3 is well defined. In the algorithm, c is a vector of n components whereas d is a vector of $|G|$ components. It is also noted that partitions that satisfy the condition in step 3.1 are not unique and that the algorithm works for any of them.

LEMMA 3.2. *Let $f \in$ **k-term-MDNF**, $h > 1$ and Y be as in $A_{\mathbf{MDNF}}$. Let v be the random variable taking values in $f^{-1}(1)$ according to the uniform distribution D^+, and put*

$$X_p = \{x_i | x_i \textit{ occurs in some predominant prime implicant of } f\},$$

and

$$X_s = \{x_i | \Pr[v_i = 1] \leq 1/2 + 1/(2^{k+2}kh)\}.$$

Then

$$\Pr[X_p \subseteq Y] \geq 1 - 1/(4h)$$

and

$$\Pr[X_s \cap Y = \phi] \geq 1 - 1/(4h).$$

Note. *X_p and X_s are uniquely determined by f, k and h, while Y determined in step 1 of the algorithm is a random variable.*

Proof. Let $x_i \in X_p$. Then, by Proposition 3.1 and the definition of a predominant prime implicant, $\Pr[v_i = 1] \geq 1/2 + 1/(2^k kh)$. Putting $p = 1/2 + 1/(2^k kh)$ and $b = 1/(2^{k+2}kh)$, we have $p(1-b) > 1/2 + 1/(2^{k+1}kh)$. Therefore, since $t_1 = R(1/2, 1/(2^{k+2}kh), 4nh) \geq R(p, b, 4nh)$, we have by Proposition 2.3

$$\begin{aligned} \Pr[x_i \notin Y] &= \Pr\left[c_i \leq \left(\frac{1}{2} + \frac{1}{2^{k+1}kh}\right) t_1\right] \\ &\leq \Pr[c_i \leq p(1-b)t_1] \\ &\leq LE(p, t_1, (1-b)pt_1) \\ &\leq \frac{1}{4nh}, \end{aligned}$$

begin
step 1:
  $c := \bar{0}; t_1 := R(1/2, 1/(2^{k+2}kh), 4nh); g := \phi;$
  for $i := 1$ **to** t_1 **do**
  begin
    v:= P-EXAMPLE;
    $c := c + v$
  end;
  $Y := \{x_j | c_j > (1/2 + 1/(2^{k+1}kh))t_1\};$
  if $k = 1$ **then**
  begin
    g:= the conjunction of variables in Y;
    goto exit
  end;
  $t_2 := \max\{\lceil \log(n) \rceil, \lceil k \log(2^{k+2}k^3h^2) \rceil \};$
  if $|Y| \leq t_2$ **then goto** step 2 **else goto** step 3;
step 2:
step 2.1: M:={monomials of the variables in Y};
step 2.2: **for** $i := 1$ **to** $L(|M|, 2^{k+|Y|}, 2h)$ **do**
  begin
    u:=N-EXAMPLE;
    for $m \in M$ **do if** $m(u) = 1$ **then** delete m from M;
  end;
step 2.3: **repeat**
    let m_0 be one of the monomials in M with the smallest size;
    $g := g \vee m_0$;
    for $m \in M$ **do if** $m \subseteq m_0$ **then** delete m from M;
  until $M = \phi$;
  goto exit;
step 3:
step 3.1: $M := \phi$;
    partition Y into $\lfloor |Y|/t_2 \rfloor$ blocks of cardinality between t_2 and $2t_2$;
step 3.2: **for** each block in the partition **do**
    construct all monomials of size $\lceil \log(2kh) \rceil$ of variables
    in the block and add them to M;
  G:={disjunctions of k monomials in M };
step 3.3: $t_3 := R((1 - 1/k)/h, 1/(2k), 4|G|h)$;
    $d := \bar{0}$;
  for $j := 1$ **to** t_3 **do**
  begin
    v:=P-EXAMPLE;
    for $g_i \in G$ **do if** $g_i(v) = 0$ **then** $d_i := d_i + 1$;
  end;
  Let d_m be the minimum value among the d_i's
  $g := g_m$;
exit:
end.

FIG. 1. *Learning algorithm* $A_{\mathbf{MDNF}}$.

where c_i is as in step 1 of algorithm $A_{\mathbf{MDNF}}$. Therefore, since there are at most n x_i's in X_p, we have

$$\begin{aligned}\Pr[X_p \not\subseteq Y] &= \Pr[\exists x_i \in X_p \; x_i \notin Y] \\ &\leq \frac{1}{4h},\end{aligned}$$

which verifies the first inequality of the lemma.

By a similar argument as above, we have

$$\Pr[X_s \cap Y \neq \phi] \leq \frac{1}{4h},$$

which verifies the second inequality of the lemma. □

LEMMA 3.3. **1-term-MDNF** *is learnable.*

Proof. The lemma follows easily from Lemma 3.2 and step 1 of $A_{\mathbf{MDNF}}$. □

LEMMA 3.4. *Let* $k \geq 1, h > 1$ *and* $f \in$ **k-term-MDNF**. *If* $A_{\mathbf{MDNF}}$ *executes step 2, then with probability at least* $1 - 1/h$, A_{MDNF} *outputs a formula* $g \in$ **k-term-MDNF** *such that* $D^+[f\Delta g] < 1/h$ *and* $D^-[f\Delta g] = 0$.

Proof. Let events E_1 and E_2 be defined as follows:

E_1 : M contains all predominant prime implicants of f for M obtained at the end of step 2.1.

E_2 : All monomials m in M with $m \not\subseteq f$ are deleted from M, i.e., $\forall m_i \in M, m_i \subseteq f$, for M obtained at the end of step 2.2.

From the first inequality of Lemma 3.2 and the fact that in step 2.1 $A_{\mathbf{MDNF}}$ constructs all monomials of variables in Y, it follows that

$$\Pr[E_1] \geq 1 - 1/(4h). \tag{3.2}$$

Let M be the set obtained in step 2.1, and let m be any monomial in M with $m \not\subseteq f$. Then for each prime implicant m' of f there exists a variable that appears in m' but not in m. Therefore, following a similar argument as in Proposition 3.1, we have

$$\begin{aligned}D^-[m] &\geq 2^{n-|Y|}/(2^k|f^{-1}(0)|) \\ &\geq 2^{n-(k+|Y|)}/2^n = 2^{-(k+|Y|)},\end{aligned}$$

because there are at most k prime implicants of f and the size of m is at most $|Y|$. Therefore, if M contains any monomial not included in f, the probability that at least one such monomial is deleted from M during the execution of the inner **for** statement in step 2.2 is at least $2^{-(k+|Y|)}$. Therefore, by the definition of $L(\cdot,\cdot,\cdot)$ we can conclude that with probability at least $1 - 1/(2h)$ it is the case that, after the **for** statement is executed $L(|M|, 2^{k+|Y|}, 2h)$ times, all monomials not included in f are deleted from M. That is,

$$\Pr[E_2] \geq 1 - 1/(2h). \tag{3.3}$$

Now assume that both E_1 and E_2 hold. And let g be obtained at the end of step 2. By E_2, we have

$$D^-[f\Delta g] = 0. \tag{3.4}$$

On the other hand, it is easy to see that E_1 implies that g includes all predominant prime implicants of f and that

$$D^+[f\Delta g] < 1/h. \tag{3.5}$$

Furthermore, noting the **for** statement in step 2.3 of A_{MDNF}, it is easy to see that any monomial of g is a prime implicant of f, which implies

$$g \in \boldsymbol{k}\textbf{-term-MDNF}. \tag{3.6}$$

Thus, by (3.2), (3.3), (3.4), (3.5), and (3.6), we have

$$\begin{aligned}&\Pr\left[g \in \boldsymbol{k}\textbf{-term-MDNF} \wedge D^{+}[f\Delta g] < 1/h \wedge D^{-}[f\Delta g] = 0\right] \\ &\geq \Pr[E_1 \wedge E_2] \geq 1 - 1/(4h) - 1/(2h) > 1 - h,\end{aligned}$$

establishing the lemma. □

LEMMA 3.5. *Let* $k \geq 2$, $h > 1$ *and* $f \in \boldsymbol{k}\textbf{-term-MDNF}$. *If* A_{MDNF} *executes step 3, then, with probability at least* $1 - 1/h$, A_{MDNF} *outputs a formula* $g \in \boldsymbol{k}\textbf{-term-MDNF}$ *such that* $D^{+}[f\Delta g] < 1/h$ *and* $D^{-}[f\Delta g] < 1/h$.

Proof. Let X_p, X_s and Y be as in Lemma 3.2. Then by Lemma 3.2,

$$\Pr[X_p \subseteq Y \wedge X_s \cap Y = \phi] \geq 1 - 1/(2h).$$

To prove the lemma, it suffices to show that, on the assumption that both $X_p \subseteq Y$ and $X_s \cap Y = \phi$ hold, algorithm A_{MDNF} outputs, with probability at least $1 - 1/(2h)$, g that is a desired approximation to target f. Let $r = \log(2^k k^2 h)$.

CLAIM 1. *If* $X_s \cap Y = \phi$ holds, *then the size of any prime implicant of* f *is at least* $|Y|/k - r$.

Proof. Assume for purposes of contradiction that there is a prime implicant m_0 of f with size $(m_0) \leq |Y|/k - r - 1$. By Proposition 3.1 and the definition of X_s, any variable not appearing in f is in X_s. Then, since $X_s \cap Y = \phi$, any variable in Y appears in f. Therefore, since there are at most k prime implicants of f, there is an $x_j \in Y$ that only appears in the prime implicants with size at least $|Y|/k$. Let f' be the disjunction of those prime implicants, with size at least $|Y|/k$, containing x_j. Then by reasoning as in the proof of Proposition 3.1, we have

$$\Pr[v_j = 1] \leq \frac{1}{2} + \frac{1}{2}\Pr[f'(v) = 1].$$

Since the size of m_0 is at most $|Y|/k - r - 1$ and the size of any prime implicant of f' is at least $|Y|/k$, we have

$$\begin{aligned}\Pr[f'(v) = 1] &= \frac{|\{v|f'(v) = 1\}|}{|\{v|f(v) = 1\}|} \\ &< \frac{|\{v|f'(v) = 1\}|}{|\{v|m_0(v) = 1\}|} \\ &\leq \frac{k2^{n-|Y|/k}}{2^{n-|Y|/k+r+1}} \\ &= \frac{k}{2^{r+1}} \\ &= \frac{1}{2^{k+1}kh}.\end{aligned}$$

Therefore,

$$\Pr[v_j = 1] \leq \frac{1}{2} + \frac{1}{2^{k+2}kh}.$$

Thus, by the definition of X_s, $x_j \in Y$ is also in X_s, contradicting the assumption $X_s \cap Y = \phi$. □

CLAIM 2. *If both $X_p \subseteq Y$ and $X_s \cap Y = \phi$ hold, then, for any predominant prime implicant m' of f, $A_{\mathbf{MDNF}}$ constructs in step 3.2 a monomial m such that $m' \subseteq m$.*

Proof. Let m' be any predominant prime implicant of f. Since $\mathrm{Var}(m') \subseteq X_p \subseteq Y$, $\mathrm{Var}(m')$ is partitioned into at most $\lfloor |Y|/t_2 \rfloor$ blocks in accordance with the partition obtained in step 3. On the other hand, by Claim 1, $|Y| > t_2$, $t_2 \geq k\log(2^{k+2}k^3h^2)$, and $r = \log(2^k k^2 h)$, we have

$$\begin{aligned} |\mathrm{Var}(m')|/\lfloor |Y|/t_2 \rfloor &\geq (|Y|/k - r)t_2/|Y| \\ &= t_2/k - t_2 r/|Y| \\ &> t_2/k - r \\ &\geq \log(2^{k+2}k^3h^2) - \log(2^k k^2 h) \\ &> \lceil \log(2kh) \rceil. \end{aligned}$$

Therefore, there exists a block (of the partition constructed in step 3) that contains at least $\lceil \log(2kh) \rceil$ variables in $\mathrm{Var}(m')$. Since algorithm $A_{\mathbf{MDNF}}$ constructs all monomials of size $\lceil \log(2kh) \rceil$ consisting of variables in the block, the claim follows. □

Let G and d_j be as in step 3 of algorithm $A_{\mathbf{MDNF}}$. Assume that $D^+[f\Delta g_j] \geq 1/h$ and that both $X_p \subseteq Y$ and $X_s \cap Y = \phi$ hold so that we can use Claim 1 and 2. By Proposition 2.3 and

$$\begin{aligned} t_3 &= R\left(\frac{1-1/k}{h}, \frac{1}{2k}, 4|G|h\right) \\ &\geq R\left(\frac{1}{h}, \frac{1}{2k}, 4|G|h\right), \end{aligned}$$

we have

$$\Pr\left[d_j > \left(1 - \frac{1}{2k}\right)\frac{t_3}{h}\right] \geq 1 - \frac{1}{4|G|h}.$$

Therefore, since there are at most $|G|$ such g_j's,

$$\Pr\left[(\forall g_j \in G)\left(D^+[f\Delta g_j] \geq \frac{1}{h} \Rightarrow d_j > \left(1 - \frac{1}{2k}\right)\frac{t_3}{h}\right)\right] \geq 1 - \frac{1}{4h}. \tag{3.7}$$

On the other hand, since G contains all combinations of k monomials constructed in step 3.2 and Claim 2 assures that each predominant prime implicant of f is included by one of the monomials constructed, there is a formula $g_i \in G$ that includes all of the predominant prime implicants of f. From this we have

$$\begin{aligned} D^+[f\Delta g_i] &< \frac{k-1}{kh} \\ &= \frac{1-1/k}{h}. \end{aligned}$$

By Proposition 2.4 and

$$\left(1 + \frac{1}{2k}\right)\left(\frac{1-1/k}{h}\right) < \left(1 - \frac{1}{2k}\right)\frac{1}{h},$$

we have

$$\begin{aligned}\Pr\left[d_i < \left(1-\frac{1}{2k}\right)\frac{t_3}{h}\right] &\geq \Pr\left[d_i < \left(1+\frac{1}{2k}\right)\left(\frac{1-1/k}{h}\right)t_3\right] \\ &\geq 1-\frac{1}{4|G|h},\end{aligned}$$

where $t_3 = R((1-1/k)/h,\ 1/(2k), 4|G|h)$. Combining this with (3.7), we have that, with probability at least $1-1/(2h) < 1-1/(4h)-1/(4|G|h)$, the following event occurs: For the g_i with $D^+[f\Delta g_i] < (1-1/k)/h$ and any $g_j \in G$ with $D^+[f\Delta g_j] \geq 1/h, d_i < d_j$ holds. This implies that algorithm $A_{\mathbf{MDNF}}$ produces g_m in **k-term-MDNF** such that $D^+[f\Delta g_m] < 1/h$. We also have

$$\begin{aligned}D^-[f\Delta g_m] &\leq \frac{|\{u|g_m(u)=1\}|}{|\{u|f(u)=0\}|} \\ &< \frac{k2^{n-\log(2kh)}}{2^{n-1}} = \frac{1}{h},\end{aligned}$$

because $|\{u|g_m(u)=1\}| \leq k2^{n-\log(2kh)}$ and $|\{u|f(u)=0\}| \geq 2^n - k2^{n-|Y|/k+r} > 2^{n-1}$ by Claim 1. □

THEOREM 3.6. *For any positive integer k,* **k-term-MDNF** *is learnable.*

Proof. The run time of step 1 is clearly Poly(n, size(f), h) for fixed k. Since $|M| = 2^{|Y|}$, $|Y| \leq \max\{\lceil\log(n)\rceil, \lceil k\log(2^{k+2}k^3h^2)\rceil\}$ and $L(|M|, 2^{k+|Y|}, 2h) \leq 2^{k+1+|Y|}(|M| + \ln(2h))$, the run time of step 2 is Poly(n, size(f), h) for fixed k. Since

$$|M| \leq (|Y|/t_2)\binom{2t_2}{\lceil\log(2kh)\rceil},$$

where $t_2 = \max\{\lceil\log(n)\rceil, \lceil k\log(2^{k+2}k^3h^2)\rceil\}$, $|Y| \leq n$, $|G| \leq |M|^k$, and $R((1-1/k)/h, 1/(2k), 4|G|h) \leq (12k^2h/(1-1/k))\ln(4|G|h)+1$, the run time of step 3 is also Poly(n, size(f), h) for fixed $k \geq 2$. Thus, by Lemma 3.3, Lemma 3.4, and Lemma 3.5, the theorem follows. □

4. Learning TH. In this section we show that **TH** is learnable.

PROPOSITION 4.1. *Let $th_{Y,t} \in$ **TH**, and let v and u be random variables taking values in $th_{Y,t}^{-1}(1)$ according to D^+ and in $th_{Y,t}^{-1}(0)$ according to D^-, respectively. Then for $i \in I(Y)$*

$$\Pr[v_i = 1] \geq t/|Y|$$

and

$$\Pr[u_i = 1] \leq (t-1)/|Y|,$$

and for $i \in I(X-Y)$,

$$\Pr[v_i = 1] = \Pr[u_i = 1] = 1/2.$$

For $Y \subseteq \{x_1, \cdots, x_n\}$ and $0 \leq k \leq |Y|$, let $S(k,Y) = \{v|v \in \{0,1\}^n, \sum_{x_j \in Y} v_j = k\}$.

PROPOSITION 4.2. *Let D^+ and D^- be the uniform distributions determined by $th_{Y,t} \in$ **TH**. If $t \leq \lfloor|Y|/2\rfloor$ then $D^-[S(t-1,Y)] \geq 2/n$, and if $t > \lfloor|Y|/2\rfloor$ then $D^+[S(t,Y)] \geq 2/n$.*

Proof. Assume $t \le \lfloor |Y|/2 \rfloor$. Since $|S(k,Y)| < |S(l,Y)|$ for $0 \le k < \ell \le \lfloor |Y|/2 \rfloor$, $|S(t-1,Y)| > |th_{y,t}^{-1}(0)|/t$. Thus, $D^-[S(t-1,Y)] > 1/t \ge 2/|Y| \ge 2/n$. For the case of $t > \lfloor |Y|/2 \rfloor$, the proposition can be proved similarly using the fact that, for $\lfloor |Y|/2 \rfloor < k < \ell \le |Y|, |S(k,Y)| > |S(l,Y)|$. □

In this section we assume that, if the set Y and the threshold t are determined, then the formula computing $th_{Y,t}$ can be constructed in polynomial time. So the key to learning **TH** is to find Y and t for $th_{Y,t} \in$ **TH**. Using Proposition 4.1 we can determine if x_j is in Y by calling P-EXAMPLE and N-EXAMPLE appropriately many times, hence set Y is obtained (step 1). Once the set Y is obtained, we can find the critical value t by counting the number of variables in Y that take value 1 in the vectors given by P-EXAMPLE and N-EXAMPLE, respectively (step 2). Proposition 4.2 assures that with high probability the critical value t so obtained is correct.

Algorithm $A_{\mathbf{TH}}$ for learning **TH** is given in Fig. 2. In fact, $A_{\mathbf{TH}}$ outputs with high probability the formula that computes a target threshold function exactly. In the algorithm error parameter $h > 1$ is given in advance, and $L(\cdot,\cdot,\cdot)$ and $R(\cdot,\cdot,\cdot)$ are as in §2. For $u, v \in \{0,1\}^n, u + v = (u_1 + v_1, \cdots, u_n + v_n)$ and $u \cdot v = (u_1 \cdot v_1, \cdots, u_n \cdot v_n)$.

begin
step 1: $c := \bar{0}$; $d := \bar{0}$; $y := \bar{0}$; $k := R(1/n, 1/(4n), 16nh)$;
  for $i := 1$ **to** k **do**
  begin
    $v :=$ P-EXAMPLE; $c := c + v$;
    $u :=$ N-EXAMPLE; $d := d + u$;
  end;
  $Y := \{x_j | c_j - d_j > k/(2n)\}$;
  for $j \in I(Y)$ **do** $y_i := 1$;
step 2: $l_p := |Y|$; $l_n := 0$;
  for $i := 1$ **to** $L(1, n/2, 4h)$ **do**
  begin
    $v :=$ P-EXAMPLE; $v := v \cdot y$;
    $l_{pe} := \sum_{1 \le j \le n} v_j$; $l_p := \min\{l_p, l_{pe}\}$;
    $u :=$ N-EXAMPLE; $u := u \cdot y$;
    $l_{ne} := \sum_{1 \le j \le n} v_j$; $l_n := \max\{l_n, l_{ne}\}$;
  end;
  if $l_p > |Y|/2$ **then** $t := l_p$ **else** $t := l_n + 1$;
step 3: Construct the formula g that computes $th_{Y,t}$ using Y and t.
end.

FIG. 2. *Learning algorithm* $A_{\mathbf{TH}}$.

THEOREM 4.3. **TH** *is learnable.*

Sketch of proof. Let $th_{Y,t}$ be a function to be learned. Throughout the proof, the set of variables Y and the threshold t are not those obtained in $A_{\mathbf{TH}}$ but those corresponding to the function to be learned. By Proposition 4.1, we can show that, with probability at least $1 - 1/(2h)$, $c_j - d_j > k/(2n)$ for any $j \in I(Y)$, and $c_j - d_j \le k/(2n)$ for any $j \in I(X - Y)$, where c_j and d_j are as in $A_{\mathbf{TH}}$ (the calculation of the probability is

straightforward application of Proposition 2.3 and Proposition 2.4, and details are left to readers). That is, $A_{\mathbf{TH}}$ computes Y correctly with probability at least $1 - 1/(2h)$. Now we assume that $A_{\mathbf{TH}}$ computes Y correctly and that $t \leq \lfloor |Y|/2 \rfloor$ (for $t > |Y|/2$, the theorem can be proved similarly). Then we can show, using Proposition 4.2 and Proposition 2.5, that $\Pr[l_p \leq |Y|/2] > 1 - 1/(4h)$ and $\Pr[l_n = t - 1] > 1 - 1/(4h)$. Therefore, $A_{\mathbf{TH}}$ computes t correctly with probability at least $1 - 1/(2h)$. Thus, since $A_{\mathbf{TH}}$ computes Y correctly with probability at least $1 - 1/(2h)$, it follows that $A_{\mathbf{TH}}$ computes Y and t correctly with probability at least $1 - 1/h$. The run time of $A_{\mathbf{TH}}$ is clearly Poly(n, size(f), h). Thus the theorem holds. □

Acknowledgment. The authors would like to thank Les Valiant for making us aware of [KMP]. We also thank the referees for careful comments. Especially, we thank one of the referees for pointing out an error in Proposition 3.1.

REFERENCES

[AV] D. ANGLUIN AND L.G. VALIANT, *Fast probabilistic algorithms for Hamiltonian circuits and matchings*, J. Comput. System Sci.,18 (1979).

[BEHW] A. BLUMER, A. EHRENFEUCHT, D. HAUSSLER, AND M.K. WARMUTH, *Classifying learnable geometric concepts with Vapnik–Chervonenkis Dimension*, Proc. 18th Annual Symposium on Theory of Computing, ACM, New York, (1986), pp. 273–282.

[C] H. CHERNOFF, *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Ann. Math. Statist., 23 (1952), pp. 493–509.

[ES] P. ERDÖS AND J. SPENCER, *Probabilistic Methods in Combinatorics*, Academic Press, New York, 1974.

[KLPV1] M. KEARNS, M. LI, L. PITT, AND L.G. VALIANT, *On the learnability of Boolean formulae*, Proc. 19th Annual Symposium on Theory of Computing, ACM (1987), pp. 185– 294.

[KLPV2] ———, *Recent results on Boolean concept learning*, Proc. 4th International Workshop on Machine Learning, Morgan Kaufmann, San Mateo, CA, 1987 , pp. 337–352.

[KMP] L. KUCERA, A. MARCHETTI-SPACCAMELA, AND M. PROTASI, *On the learnability of DNF formulae*, ICALP (1988), pp. 347–361.

[KV] M. KEARNS AND L.G. VALIANT, *Cryptographic limitations on learning Boolean formulae and finite automata*, Proc. 21th Annual Symposium on Theory of Computing, ACM (1989), pp. 433–441.

[N] B.K. NATARAJAN, *On learning Boolean functions*, Proc. 19th Annual Symposium on Theory of Computing, ACM (1987), pp. 296–304.

[OM] T. OHGURO AND A. MARUOKA, *A learning algorithm for monotone k-term DNF*, in FUJITSU IIAS-SIS Workshop on Computational Learning Theory '89, (1989).

[PV] L. PITT AND L.G. VALIANT, *Computational limitation on learning from examples*, J. Assoc. Comput. Mach., 35 (1988), pp. 965–984.

[V1] L.G. VALIANT, *A theory of the learnable*, Comm. ACM, 27 (1984), pp. 1134–1142.

[V2] ———, *Deductive learning*, Tech. Report, TR–06–84, Harvard University, Cambridge, MA, 1984.

SIAM J. COMPUT.
Vol. 21, No. 4, pp. 601–638, August 1992
© 1992 Society for Industrial and Applied Mathematics
001

SHORTEST PATHS HELP SOLVE GEOMETRIC OPTIMIZATION PROBLEMS IN PLANAR REGIONS*

ELEFTERIOS A. MELISSARATOS† AND DIANE L. SOUVAINE†

Abstract. The goal of this paper is to show that the concept of the shortest path inside a polygonal region contributes to the design of efficient algorithms for certain geometric optimization problems involving simple polygons: computing optimum separators, maximum area or perimeter-inscribed triangles, a minimum area circumscribed concave quadrilateral, or a maximum area contained triangle. The structure for the algorithms presented is as follows: (a) decompose the initial problem into a low-degree polynomial number of optimization problems; (b) solve each individual subproblem in constant time using standard methods of calculus, basic methods of numerical analysis, or linear programming. These same optimization techniques can be applied to splinegons (curved polygons). First a decomposition technique for curved polygons is developed; this technique is substituted for triangulation in creating equally efficient curved versions of the algorithms for the shortest-path tree, ray-shooting, and two-point shortest path problems. The maximum area or perimeter inscribed triangle problem, the minimum area circumscribed concave quadrilateral problem and maximum area contained triangle problem have applications to robotics and stock-cutting. The results of this paper appear in E. A. Melissaratos's Ph.D. thesis [*Mesh Generation and Geometric Optimization*, Rutgers University, New Brunswick, NJ, 1991].

Key words. robotics, stock-cutting, computational geometry, enclosure problems, inclusion problems, separators, geometric optimization, shortest paths, visibility, simple polygons, splinegons

AMS(MOS) subject classifications. 68U05, 68Q25, 68Q20, 51M15, 52A27

1. Introduction. The linear-time algorithm for computing the lengths of the shortest paths inside a triangulated simple polygon from a designated start vertex by Guibas et al. [26] provides a useful tool in developing efficient polygon algorithms for a class of geometric optimization problems. Although our main results refer to the polygon case of the optimization problems, we extend our results to the curvilinear case also. Souvaine and Dobkin have recently argued that, wherever possible, new results should be presented for polygons and curved polygons simultaneously [18]. To make these extensions feasible, we need to develop algorithms for shortest paths and visibility problems in curvilinear objects. Unfortunately, shortest paths and visibility represent an area in which little work has been done on curved polygons.

In order to express our results on the optimization problems in the most general terms possible, we begin by focusing on decompositions, shortest paths, and visibility in splinegons (curved polygons in which the region bounded by each curved edge and the line segment joining its endpoints is always convex [18]). The polygonal shortest path and visibility algorithms all require a triangulated polygon. Triangulation, however, is not a viable method on splinegons: it may require adding additional vertices, both on the boundary and in the interior; furthermore, curved triangles are not necessarily convex [18], [19], [41]. By substituting a new *bounded degree decomposition* that is linear-time equivalent to triangulation, we generate equally efficient curved versions of the polygon algorithms for creating shortest paths and factor graphs and for solving visibility from an edge, ray-shooting, and two-point shortest paths.

We then use shortest paths to design algorithms for several types of geometric optimization problems on both polygons and splinegons: separators, inscribed triangles,

*Received by the editors July 30, 1990; accepted for publication (in revised form) June 4, 1991. This research was supported in part by National Science Foundation grant CCR-88-03549 and by Center for Discrete Mathematics and Theoretical Computer Science under National Science Foundation grant STC-88-09648. Preliminary reports on this work have appeared in the Proceedings of the 6th ACM Symposium on Computational Geometry and in the Proceedings of the 2nd Canadian Conference on Computational Geometry.

†Department of Computer Science, Rutgers University, New Brunswick, New Jersey 08903.

circumscribed concave quadrilaterals, and contained triangles.

Separators. If two points x and y lie on the boundary of a simple polygon or splinegon P and define a directed line segment $xy \subseteq P$ that separates P into two sets, P_L and P_R, then xy is called a *separator*.

> **Minimum length.** The areas of P_L and P_R are defined by constants K_L and K_R. Find a separator of minimum length such that the ratio of the areas of P_L and P_R remains equal to a given constant.
>
> **Minimum sum of ratios.** Find a separator that minimizes the sum of the ratio of the area of P_L to the square of its perimeter and the ratio of the area of P_R to the square of its perimeter.

Inscribed triangles. Given a simple polygon or splinegon P, a triangle T where $T \subseteq P$ and the vertices of T lie on the boundary of P is called an *inscribed triangle*.

> **Maximum area.** Find the inscribed triangle of maximum area.
>
> **Maximum perimeter.** Find the inscribed triangle of maximum perimeter.
>
> **Constrained maximum area/perimeter.** Find a maximum area/perimeter inscribed triangle with one edge of given length.

Circumscribed quadrilateral. Given a simple polygon or splinegon P, a quadrilateral Q where $P \subseteq Q$ and all four sides of Q intersect the boundary of P, is called a *circumscribed quadrilateral*.

> **Minimum area concave.** Find the circumscribed concave quadrilateral of minimum area.

Contained triangle. Given a simple polygon or splinegon P, a triangle T where $T \subseteq P$ is called a *contained triangle*.

> **Maximum area.** Find the contained triangle of maximum area.

In each case, we find the global optimum by using shortest paths to decompose the optimization problem into a low-degree polynomial number of simple continuous optimization problems; each problem is solved in $O(1)$ arithmetic operations by using the methods of calculus analytically, standard methods of numerical analysis, or linear programming, and computing the optimum of all the local optima. For polygons, we solve the separator problems in $O(n^2)$ time, the inscribed triangle problems in $O(n^3)$ time, and the contained triangle problem in $O(n^4)$ time, all in linear space. Subsequently, we combine our techniques with Hershberger's output-sensitive visibility graph technique [29], to create modified algorithms for the area separator and maximum area or perimeter inscribed triangle problems, which run in $O(m)$ and $O(n^2 + nm)$ time, respectively, where m is the size of the visibility graph of P. The quadrilateral problem can be solved either in $O(n_c^2 n_p^2)$ time and $O(n)$ space or in $O(n_c^2(n_p + k))$ time and $O(n + k)$ space, where n_c is the number of vertices of the convex hull of P, $n_p = n - n_c$, and k is an instance-dependent parameter that ranges between $O(n_p)$ and $O(n_p^2)$. Although some of the algorithms for curvilinear objects obtain the same asymptotic complexity as their polygonal counterparts, others do not: the splinegon separator algorithm runs in $O(n^2)$ time; the inscribed triangle algorithm in $O(n^4)$. We conjecture that some curvilinear problems are inherently more difficult than their polygonal counterparts.

The decomposition step used to solve both the inscribed triangle problems and the contained triangle problem focuses on a new type of computationally tractable polygon (respectively, splinegon), the *fan-shaped polygon* (respectively, *fan-shaped splinegon*). Every triangle inscribed (contained) in a simple polygon or splinegon P is also inscribed (contained) in a fan-shaped polygon or splinegon $P' \subseteq P$. We expect that the fan-shaped polygon/splinegon will become a useful tool in other applications as well.

The next two paragraphs recite some of the history of the polygon version of these problems. Lisper [32] posed the first separator problem, citing applications in solid modeling and graph cutting. A linear algorithm exists for convex polygons [36]. Chang posed the second separator problem [10], claiming applications in finite element analysis. Aggarwal [1] posed both the contained triangle problem and the concave circumscribed quadrilateral problem. Previously, Chang [10] and Chang and Yap [9] had posed the contained triangle problem as an open problem. There are numerous results for related problems. The Klee–Laskowski bound of $O(n \log^2 n)$ time for computing the minimum area triangle containing a convex n-gon time [30] was improved to linear time by O' Rourke et al. [37]. Finding the minimum area circumscribing k-gon of a convex n-gon was solved first in $O(n^2 \log k \log n)$ time [2], next in $O(n^2 \log k)$ time [3], and finally in $O(nk + n \log n)$ by Aggarwal and Park [4]. DePano's bound of $O(n^3)$ time for computing the minimum perimeter triangle circumscribing a convex n-gon [15] was improved by Aggarwal and Park to $O(n \log n)$ time [4]. Note that any convex k-gon circumscribing a simple polygon P also circumscribes the convex hull of P.

Many researchers have studied inclusion problems. Dobkin and Snyder [17] presented a linear-time algorithm for computing the minimum area triangle inscribed in a convex polygon of n vertices. Boyce et al. [5] computed the maximum area or perimeter convex k-gon inside a convex n-gon in $O(kn \log n + n \log^2 n)$ time. Aggarwal et al. [3] improved the bound to $O(kn + n \log n)$. Chang and Yap solved the general problem of finding the maximum area (perimeter) convex polygon contained within a given simple polygon P in $O(n^7)$ time (respectively, $O(n^6)$ time) and $O(n^5)$ space [9], [10]. DePano et al. [16] gave an $O(n^3)$ algorithm for a maximum area equilateral triangle contained in a simple polygon. This result can be improved using a recent result of Chew and Kedem [14] for the problem of placing the largest similar copy of a convex k-gon in an arbitrary polygonal environment. Fortune [22] solved the problem of placing the largest homothetic copy of a k-gon in a simple polygon in $O(kn \log kn)$ time. Some recent research has focused on simultaneous inner and outer approximation of convex polygons by a pair of rectangles [39] or by a pair of similar triangles [21].

We have recently learned of some independent work on shortest paths and visibility in curved regions. Many interesting, nonalgorithmic, properties of shortest paths inside curvilinear regions appear in [7], [8]. Furthermore, Bourgin and Howe [6] provide algorithms for shortest paths between two fixed points in a Jordan region that run in $O(nk)$ time, where n is the number of distinct sections of the boundary (i.e., the number of vertices of the boundary) of the region and k is the number of the vertices on the shortest path. Our algorithm computing the lengths of the shortest paths from a fixed point to *all* the vertices of the boundary of the region runs in $O(n)$ time. When restricted to computing the shortest path between two fixed points, our algorithm would use $O(n)$ time for the length computation or $O(n + k)$ time for computing the actual path, where k is the number of the vertices of the shortest path.

In the next section, §2, we review polygon shortest path and visibility results, develop the corresponding splinegon versions, and establish the notation to be used in the paper. Sections 2.2–2.5 are long and detailed and may be disregarded by readers primarily

interested in the polygonal versions of the optimization results. Sections 3–6 examine each of the optimization problems in turn. Preliminary results of this paper appear in [34], [35]. The results of this paper will appear also in Melissaratos's forthcoming thesis, [33].

2. Shortest paths and visibility in polygons and splinegons.

2.1. Shortest paths in simple polygons. In the next few paragraphs we establish our conventions and review necessary facts and definitions. A simple polygon or splinegon P has n edges represented by the integers $1, 2, \cdots, n$ in clockwise order, and edge j has endpoints p_j and p_{j+1}; whenever a subset of polygon or splinegon vertices (edges) are identified by uppercase (lowercase) letters, alphabetic order implies clockwise order around P; a line l is *tangent* to a polygonal chain C if it intersects the chain in one or more points and C lies entirely in one of the halfplanes defined by l. A point $x \in P$ is *visible* from an edge i of P if there exists a point y on i such that $xy \subseteq P$. Two edges i, j of P are *visible* from each other if and only if there exist at least two points x, y on i, j, respectively, such that $xy \subseteq P$. The set of points $x \in P$, which are visible from an edge i, form the *visibility polygon* or *visibility splinegon* of P from edge i. If two edges i and j of P are visible from each other, the set of points of j that are visible from i form the *visible part of j with respect to i.*

In [26] the authors describe a linear-time and space algorithm for finding the shortest paths from a point v inside or on the boundary of P to all its vertices, if P represents a triangulated simple polygon. The union of these paths forms a tree called *the shortest path tree with respect to source v*, or just *the shortest path tree*, if v is understood. The shortest path algorithm applied to a triangulated simple polygon P at a designated start vertex v produces a subdivision where each region corresponds to a *funnel* based on some polygon edge i, denoted $F_v(i)$. (See Fig. 1(a).) Extending the edges of each funnel up to their intersection with the funnel's base produces the *extended shortest path tree* which induces a refined subdivision of P where every region is a triangle. This configuration is called the *shortest path map with respect to v* or simply the *shortest path map*, if v is understood. The extended shortest path tree from a vertex v subdivides each edge i of P into *elementary segments*. This set of elementary segments on edge i is denoted by $S_v(i)$ and its size by s_i^v. The union of all $S_v(i)$ over all edges i of P is called the *trace of* v. The closest vertex to x on the shortest path from v to x is called the *anchor* of x with respect to v and is denoted by $anchor^v(x)$. A fundamental property of each shortest path map of a polygon is that all points x in a particular region of the shortest path map have the same anchor (see Fig. 1(b)) [26], [29].

One can compute the visible parts of a given edge i from every other edge of the polygon, as well as the visible parts of every edge from i, in $O(n)$ time and space using the shortest path algorithm. If edges i, j of polygon P are visible from each other, then the shortest paths from p_{j+1} to p_i ($SP_{p_{j+1}}(p_i)$) and from p_{i+1} to p_j ($SP_{p_{i+1}}(p_j)$) are inward disjoint convex chains. The region bounded by the above chains and i and j is called an *hourglass* and denoted $H_{i,j}$ [26].

Assuming that polygon P is triangulated, the shortest path algorithm of [26] from a vertex s of P proceeds as follows. Assume without loss of generality that s lies on only one triangle. The computation corresponds to a preorder traversal of the binary tree with one node for each triangle, with an edge joining two nodes whose triangles share an edge, and with the triangle containing s as the root. The algorithm maintains the invariant that all funnels for polygon edges belonging to processed triangles and edges of current triangles (nodes) have been computed and are stored in finger search trees.

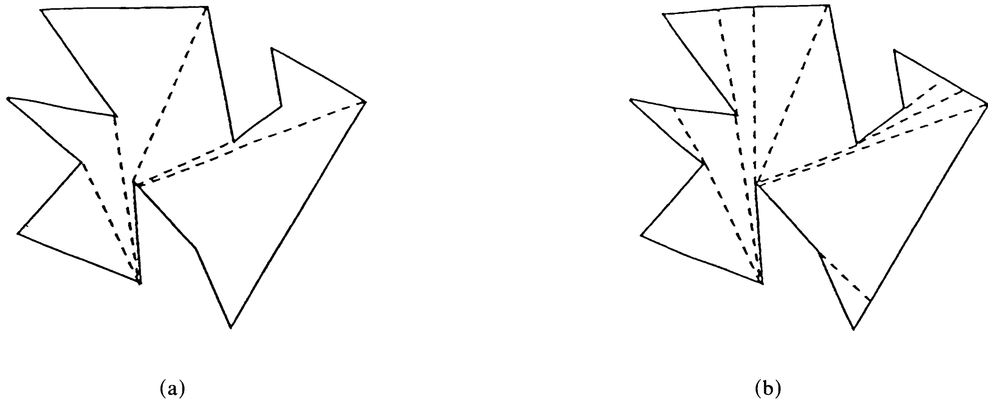


FIG. 1. (a) *Shortest-path tree;* (b) *shortest-path map.*

This statement is trivially true at the outset, when we process the triangle that contains s. It has only one edge interior to P, an active edge. Go through that edge, splitting its funnel by computing a tangent from the new vertex to one of the chains of the funnel; the funnels for each of the other two edges of the next triangle are formed in this way. If those two edges both lie on the boundary of P, then this triangle is a leaf. If both are interior to P, then this triangle has two children. Otherwise, this triangle has one child [26]. The use of the triangulation of the polygon as well as of the funnels, in the computation of shortest paths inside simple polygons, appeared first in a paper by Lee and Preparata [31]. Lee and Preparata present an algorithm to compute the length of the shortest path between two fixed points inside a simple polygon. The difference is in the following two aspects: (a) funnel representation (Lee and Preparata represent the funnels as linked lists and not as finger search trees), and (b) when a funnel is split the algorithm of [31] recurs to only one of the funnels, although the algorithm of [26] may recur to both split funnels.

2.2. Bounded degree decomposition of splinegons. To extend the polygonal shortest path algorithm [26] to work for splinegons, we first need to find an acceptable substitute for triangulation, since Dobkin et al. [19] have shown triangulation of splinegons to be infeasible. One candidate decomposition is the horizontal visibility map which would decompose the splinegon into horizontal trapezoids (with curved sides). Fortunately, the Tarjan–Van Wyk algorithm [42] is applicable to splinegons, provided that the edges are monotone to at least one of the axes [19], as is the new linear-time Chazelle algorithm [11]. Both algorithms produce a linear number of new vertices. Ordinarily, there will be at most one interior vertex per trapezoid edge. However, in some applications, several vertices may have the same y-coordinate, producing an arbitrary number of vertices within a base of a trapezoid. Thus in theory some trapezoids could have an unlimited number of neighbors. Our goal is to refine the curvilinear trapezoids so that every component has at most three neighbors so that the dual of the decomposition is a binary tree, a key characteristic of polygonal triangulations.

To guarantee that this decomposition is sufficiently general, we need to verify that even in these degenerate cases, the decomposition can be accomplished by adding new vertices only on splinegon boundaries. We call our decomposition of a simple splinegon, into components with at most four sides and with at most three neighbors, the *bounded*

degree decomposition. To begin, we preprocess the edges of the splinegon such that each edge is monotone with respect to *both* the x and the y axes, i.e., insert the extrema of each splinegon edge with respect to either axes as a new vertex, without duplicating edge endpoints. The convexity of the splinegon edges means that each edge can have at most one minimum and at most one maximum relative to each axis, and, consequently, the additional number of edges or vertices is at most $4n$. If constant time suffices to compute the extrema of any edge, then this preprocessing uses $O(n)$ time.

Next, we separate the set of curved trapezoids and triangles of the horizontal visibility decomposition into three groups: *Group I* contains those that have two side edges concave; *Group II*, those with one side concave and the other convex; and *Group III*, the ones with both sides convex. Remember that, given our preprocessing, all curved edges are both x-monotone and y-monotone. We focus primarily on *Group I*, since the techniques we develop for that group can clearly be applied for the second and third groups also, although simpler procedures for those groups would suffice. We classify each trapezoid $ABCD$ of *Group I* with bases AB, CD and side edges AD, BC as having one of three subtypes, by comparing the projections of each of AB, CD to a line parallel to both:

Type 1: The projections intersect but neither is contained in the other.
Type 2: The projection of AB is contained in the projection of CD.
Type 3: The projections have empty intersection.

For all three types, call a vertex interior to a base that vertically projects onto a curved side of the trapezoid an $a - point$, and add its projection as a $new - vertex$. All other vertices interior to bases are called $b - points$ (Fig. 2).

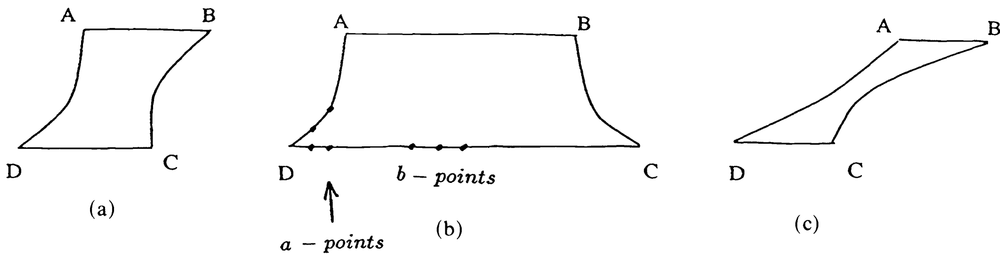


FIG. 2. (a) *Type* 1 *trapezoid;* (b) *Type* 2 *trapezoid;* (c) *Type* 3 *trapezoid.*

If $ABCD$ belongs to Type 1, then connect A (respectively, C) to any $b - points$ between C (respectively, A) and E (respectively, F) where E (respectively, F) is the projection of A (respectively, C) onto CD (respectively, AB); if there are no $b - points$ then insert the diagonal AC. Connect each new point on the splinegon edge DA (respectively, BC) generated by an $a - point$ on DC (respectively, BA) to the next vertex on that edge by a diagonal (Fig. 3(a)).

If $ABCD$ belongs to Type 2, let E, F be the projections of B, A on DC, respectively, such that E, F lie in segment DC. If there exists at least one $b - point$ on CD, then the decomposition is done as in Fig. 3(b). If there are no $b - points$ on CD and there are on AB, then let G (respectively, H) be the vertical projection on AD (respectively, BC) of the last $a - point$ as we move right (respectively, left) from D (respectively, C). Assume that the y-coordinate of G is larger than that of H. Let I represent the horizontal projection of G on BC. Then what remains is to decompose the trapezoid $ABIG$ (provided it has $b - points$ on AB), by connecting G to all $b - points$ on AB (Fig.

3(c)). If $ABCD$ is of Type 3, then there are no $b-points$ and the decomposition is given in Fig. 3(d).

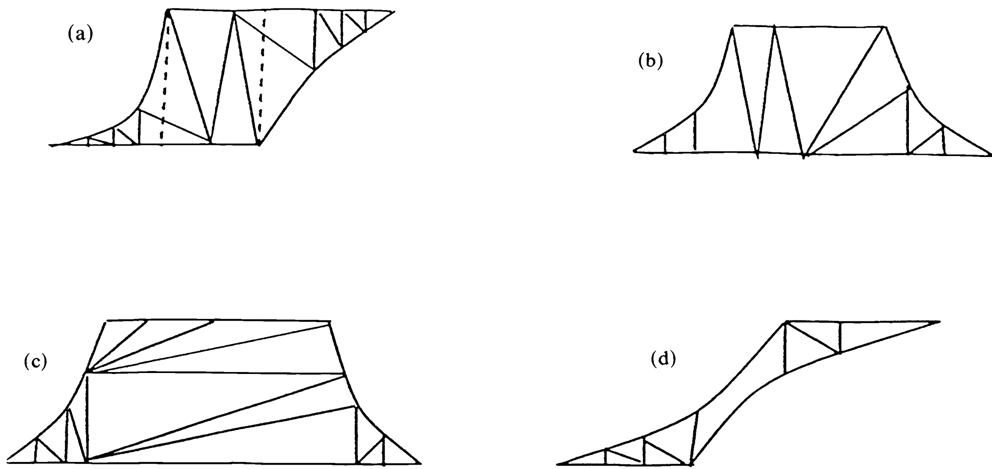


FIG. 3. *Refinement of degenerate trapezoids:* (a) *Type* 1; (b) *and* (c) *Type* 2; (d) *Type* 3.

Finally, where possible, all remaining quadrilaterals are triangulated using a diagonal. Given that splinegon edges are monotone with respect to both the x and y axes, the only class of quadrilaterals that cannot be triangulated by a diagonal consists of trapezoids of Type 3: trapezoids with concave-out curved side edges on the boundary of the splinegon and parallel bases with disjoint x-intervals. These quadrilaterals have exactly two neighbors.

THEOREM 2.1. *Any simple splinegon can be decomposed in such a way that each component has at most three neighbors, in the same asymptotic time complexity as triangulating a simple polygon.*

Proof. The algorithm described above first computes the horizontal visibility information and then spends at most constant time per vertex refining the decomposition. Fournier and Montuno have proved that triangulating a polygon is linear-time equivalent to computing horizontal visibility information [23]. □

2.3. Shortest paths in splinegons. Minor revisions allow the polygonal shortest path algorithm [26] to work for curved polygons, also known as *splinegons*, in comparable time and space bounds. The curved algorithm maintains the corresponding invariant that all funnels for polygon edges belonging to processed and current components of the bounded degree decomposition have been computed and are stored in finger search trees, but there are several notable differences. A convex chain of a funnel is not necessarily made of straight line segments but is rather a concatenation of straight line segments and convex curved segments. If the shortest path map is formed by extending the straight line segments of the funnels and the tangents at the endpoints of the curved segments up to the intersection of the corresponding splinegon edge, for a point x moving within a single component of the shortest path map, $anchor^s(x)$ is not a constant function as in the polygon case. Instead, $anchor^s(x)$ varies over a particular convex section of a single curved edge of the splinegon boundary. Although in the polygon case the funnel splitting operation does not create new vertices on the boundary of the polygon, with splinegons, a new vertex may be created either in the funnel or in the boundary of a new component or on both. All of these differences can be accommodated.

THEOREM 2.2. *The shortest path tree inside a simple splinegon with a designated root can be computed in $O(n)$ time, given the bounded degree decomposition.*

Proof. The main step of the polygon algorithm is as follows: given a funnel and a triangle, with one of its sides coincident with the funnel base, split the funnel into at most two funnels, which have the other two edges of the triangle as bases. The funnel algorithm needs to be revised to accommodate different types of regions: triangles with one or more curved edges; straight-edged triangles; and quadrilaterals with two curved edges. Clearly, a region with two children must be a straight-edged triangle. Thus, only regions with at most one child need different processing. If the total contribution of the one-child components to the complexity of the algorithm remains $O(n)$, then the recursive formula used in [26] to prove the linearity of the entire algorithm still applies. There are two main changes to make. One is that splitting the funnels may involve computing tangents from a point to a curve or between a pair of curved edges. But we may assume that each curved operation requires constant time [18], there is no asymptotic penalty.

More importantly, at a node corresponding to a quadrilateral, there are two splitting points rather than just one (see Fig. 4). Let t_1, t_2 be the two splitting points of the current funnel on base AB. Let n_1, n_2, n_3 be the number of funnel vertices between A and t_1, t_1 and t_2, t_2 and B, respectively. In Fig. 4, the current funnel is split into three funnels: the first has apex t_1 and consists of the convex chain from A to t_1 together with the common tangent t_1q_1 and has the convex segment Aq_1 as base; the second funnel has apex t_2 and consists of the splinegon boundary segment Dq_1 followed by the common tangent t_1q_1, by the convex subchain t_1t_2, the common tangent t_2q_2, and by the splinegon boundary segment q_2C; the third funnel has apex s, has the splinegon boundary segment q_2B as base, and consists of the common tangent t_2q_2 followed by the convex subchain t_2s and the convex subchain sB. The first and the third of these funnels are not processed further, since their bases lie on the splinegon boundary. Thus the vertices of the original funnel between A and t_1 and between t_2 and B will not be used again by the algorithm. Since the funnels are represented by finger trees, the first splitting and tangency operation takes time $O(\min(\log(n_1), \log(n_2+n_3)))$ and the second $O(\min(\log(n_2), \log(n_3)))$. But the first is $O(n_1)$ and the second $O(n_3)$. Therefore the total complexity for that case is $O(n_1+n_3)$. But n_1+n_3 is the number of "dead" vertices of the funnel (i.e., the vertices that are not going to be processed further). Therefore summing over all zero or one child cases gives $O(n)$. □

We can also compute the shortest path tree inside a simple splinegon directly from the horizontal visibility decomposition without computing the bounded degree decomposition, producing an alternate proof.

Proof. At any time, we consider the current funnel and a trapezoidal component of which the parallel sides may contain many splinegon vertices (Fig. 5). The shortest paths from the apex α of the funnel to $v_1, v_2, \cdots, v_k$ and to $w_1, w_2, \cdots, w_l$ create new funnels with bases v_iv_{i+1} for $i = 1, \cdots, k$ and w_jw_{j+1} for $j = 1, \cdots, l$. It suffices to solve the following subproblem: given a funnel with apex α, base bc and convex chains F_1 and F_2, a trapezoid $bcde$ and a point x on de, find the shortest path from α to x that lies inside the area defined by the funnel and the trapezoid. Call the curved sides of the trapezoid C_1 and C_2. Consider the tangent from x to the funnel. Let t be the corresponding tangent point. Without loss of generality, assume t lies on F_1. We have the following cases (Fig. 6).

1. xt does not intersect any of the C_1 or C_2. Then the shortest path from α to x is the concatenation of the part of F_1 from α to t and the segment xt.

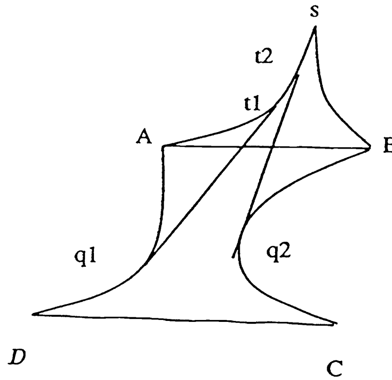


FIG. 4. *Funnel vertices between A and t_1 and vertices between t_2 and B will not be used again by the algorithm.*

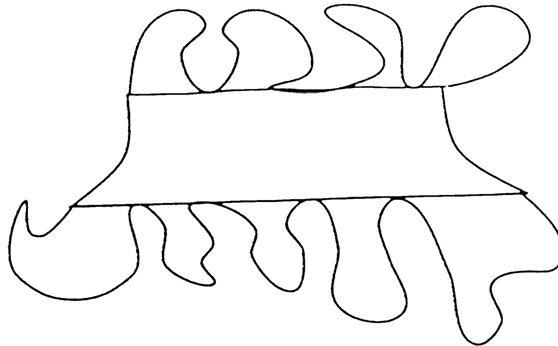


FIG. 5. *A trapezoidal component with many splinegon vertices interior to the parallel sides.*

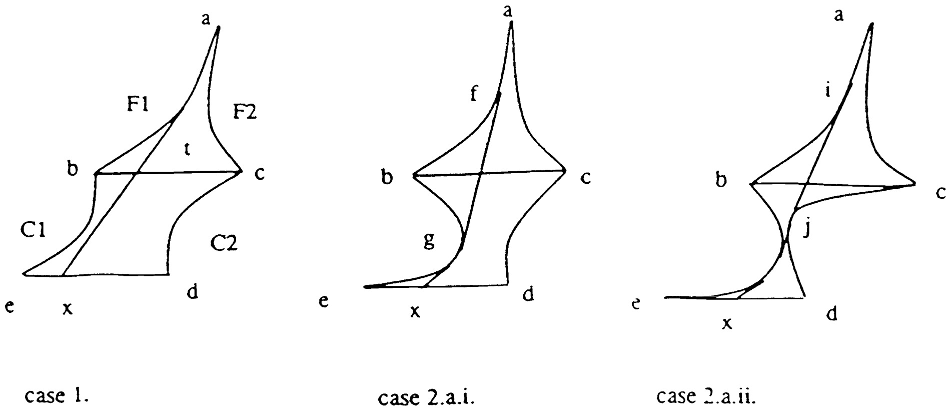


FIG. 6. *Several cases of funnel splitting.*

2. xt intersects only one of the C_1 or C_2.
   (a) xt intersects C_1. Let fg be the common outer tangent of F_1 and C_1, where f is on F_1 and g is on F_2.
      (i) fg does not intersect C_2. Then let xh be the tangent from x to C_1. Then the shortest path is af, fg, gh, hx.

(ii) fg intersects C_2. Then let ij be the inner common tangent of F_1 and C_2 and kl be the inner common tangent of C_1, C_2. Then the shortest path is ai, ij, jk, kl, lh, hx.

(b) xt intersects C_2. Use the same steps as in case 2(a), but let fg be the inner common tangent of F_1 and C_2.

3. xt intersects both C_1 and C_2.
   (a) As we move from x to t, xt intersects first C_1 and then C_2. Then let fg be the inner common tangent of F_1, C_2, xh be the tangent from x to C_1, and ij the inner common tangent of C_1 and C_2. Then the shortest path is af, fg, gi, ij, jh, hx.
   (b) As we move from x to t, xt intersects first C_2 and then C_1. Then let fg be the outer common tangent of F_1, C_1, ij the inner common tangent of C_1, C_2, and xh be the tangent from x to C_2. Then the shortest path is af, fg, gi, ij, jh, hx.

We use this funnel-splitting operation recursively. Following the above approach, a funnel may be split into more than two subfunnels, thus we cannot apply the recurrence formula as in the two-way splitting. But this multiway splitting can be simulated by two-way splittings. The horizontal visibility decomposition of the splinegon is a planar subdivision where its dual is a tree, not necessarily binary. Make that tree a rooted tree, choosing arbitrarily any node as the root. Thus in our rooted tree every node except the root has indegree equal to one. We now apply a well-known transformation which converts any tree to a binary one. Assume that node v has parent node u and children nodes w_1, w_2, w_3, w_4. The transformation constructs a binary tree with root v and leaves w_1, w_2, w_3, w_4 (Fig. 7). Then for the complexity analysis the same arguments can apply since we have to work with a binary tree. We must note that the above transformation has nothing to do with the implementation of the algorithm. It is useful only for the complexity analysis. □

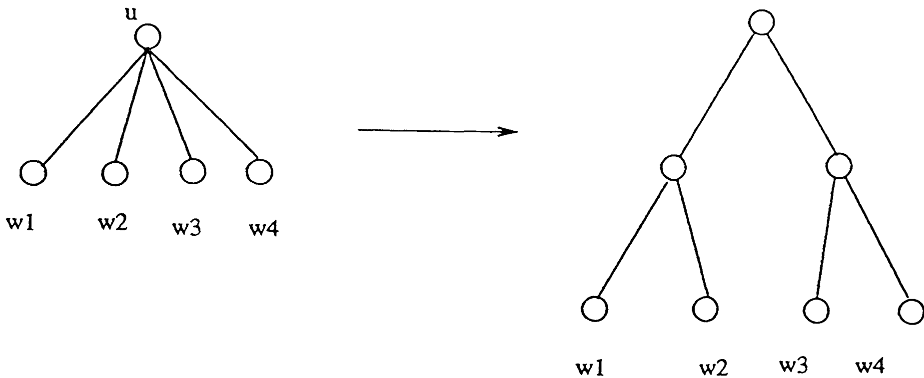


FIG. 7. *Any arbitrary tree may be converted to a binary tree.*

The second approach uses the horizontal visibility decomposition directly instead of the bounded degree decomposition and applies multiway splitting of the funnels instead of the original two-way splitting. The two approaches are equivalent from the point of view of asymptotic time complexity but not in practice. Although to get the bounded degree decomposition requires only linear time, the constant may represent computation of intersections of higher degree curves. The second approach avoids these computa-

tions, but requires a somewhat complicated proof that multiway splitting does not affect the linearity of the algorithm. It might seem, therefore, that the concept of the bounded degree decomposition is unnecessary. Some visibility problems do not present this option. Although computing the visibility splinegon from an edge depends only on shortest paths, however they are obtained, problems like ray-shooting and the two-point shortest path problem depend on an augmented balanced decomposition tree such as the *factor graph*, which is computable from the bounded degree decomposition.

2.4. Factor graphs of splinegons. A triangulation of a polygon can be converted in linear time to a balanced binary decomposition tree in which each node corresponds to a subpolygon P and to a diagonal d, which divides P so that neither of the two children subpolygons P_L and P_R contains more than $\frac{2}{3}$ of the triangles of P; d roughly bisects P [26]. As all of the diagonals in the bounded degree decomposition of a splinegon S are straight segments, this algorithm extends directly to splinegons.

Assume that S is the initial polygon or splinegon and that we are given a balanced decomposition tree for S. Let S_l be a polygon or splinegon at level l in the decomposition tree, and let d_l be the roughly bisecting diagonal of S_l. The boundary of S_l consists of some edges of S and some diagonals. The *factor graph* [13] has edges between d_l and the bounding diagonals of S_l; in other words, edges of the factor graph correspond to pairs of bisecting diagonals. Some visibility applications need an augmented factor graph in which each edge is equipped with a representation of the hourglass corresponding to that pair of diagonals. The bottom-up polygon algorithm for creating the (augmented) factor graph extends easily to splinegons. Beginning with the balanced decomposition tree, construct the trivial hourglasses for regions represented by the leaves. Now assume that all hourglass computation up to level k has been completed. Thus, for any splinegon component in levels 1 to k the hourglasses between any pair of bounding diagonals have been computed. To proceed to level $k+1$, "delete" all the diagonals at level k and compute the hourglasses between any bounding diagonal of the left component and any bounding diagonal of the right component by trimming and then concatenating the two hourglasses at level k. The hourglasses may now contain both straight edges and portions of curved edges so that the trimming operation may involve computing tangents to curved edges, but the essential procedure is unchanged.

Since representing an hourglass explicitly uses $O(n)$ space, the augmented factor graph could use $O(n^2)$ space overall. By keeping each edge of an hourglass only at the highest level in which it appears in the tree, Chazelle and Guibas [13] demonstrated that the augmented factor graph could be designed to have the following properties: the augmented factor graph has size $O(n)$, each node has degree $O(\log n)$, and the graph can be constructed from the decomposition of the polygon in $O(n)$ time.

THEOREM 2.3. *The factor graph and the augmented factor graph of a simple splinegon S can be computed in $O(n)$ time and space, given the bounded degree decomposition.*

2.5. Visibility results for splinegons. In this section, we consider the following three problems:

1. *Visibility from an edge:* Given an edge j of splinegon S, find the points x on the boundary of S for which there exists at least one point y on j such that $xy \subseteq S$.
2. *Ray-shooting:* Given a simple splinegon S, a query point q, and a ray passing through q, find the first intersection of the ray with the splinegon.
3. *Two-point shortest path problem:* Given a simple splinegon S, preprocess it in order to construct a data structure such that given any two query points p and q the length and the shortest path itself can be computed efficiently.

Each of them can be solved efficiently using either shortest paths or factor graphs. The last two, however, also use planar point location. To preprocess S for this purpose, first construct the convex hull of S, $CH(S)$, in linear time [18], [40]. Given the bounded degree decomposition both of S and of the pockets identified in the process of computing the convex hull, a *layered dag* can be constructed in linear time, allowing the location of a query point in a component of the decomposition to be determined in logarithmic time [20].

THEOREM 2.4. *Given the horizontal visibility decomposition of S, computing the part of the boundary of a simple splinegon S of n vertices that is visible from an edge requires* $O(n)$ *time.*

Proof. The linear-time polygon algorithm uses the fact that if edges i, j are visible from each other, then the shortest paths from p_{j+1} to p_i ($SP_{p_{j+1}}(p_i)$) and from p_{i+1} to p_j ($SP_{p_{i+1}}(p_j)$) are inward disjoint convex chains [26]. For splinegons, this fact does not hold (Fig. 8). We present a new method based on local computations for computing the visibility of an edge in either a polygon or a splinegon.

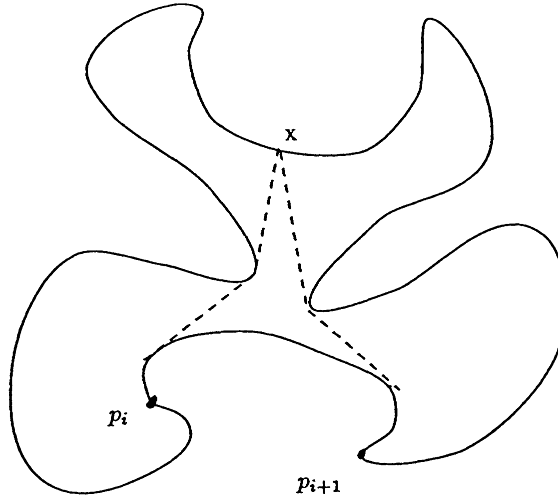


FIG. 8. *Shortest paths from edge endpoints* p_i *and* p_{i+1} *are not inward convex chains.*

To compute the visible region from edge j, find the shortest path maps from p_j and p_{j+1}, respectively. Merge $trace(p_j)$ and $trace(p_{j+1})$ into a linear-sized subdivision M. If x moves along an elementary segment I of M, the anchors of x with respect to the endpoints of j remain unchanged. Thus we can unambiguously refer to $anchor^{p_j}(I)$ and $anchor^{p_{j+1}}(I)$. For each I of M, perform the following simple test: if $anchor^{p_{j+1}}(I) <> anchor^{p_j}(I)$, then for every point x on I, x is visible from edge j; call such a segment I a *valid* segment. Merge adjacent valid segments and then report the results. □

THEOREM 2.5. *Given the bounded degree decomposition, the factor graph, and the layered dag of a simple splinegon S and all of its pockets, a query point q and a ray passing through q, the first intersection of the ray with the splinegon can be reported in* $O(\log n)$ *time.*

Proof. Locate the query point q in the decomposition using the layered dag [20]. If q is outside $CH(S)$, then determine the convex hull edge first crossed by the shooting ray in logarithmic time [12], [18]. If it is an edge of S, report it. If not, perform ray shooting as described below in the pocket having that edge as a lid, a new splinegon. As in [13], if q lies within S, find the diagonal crossed by the shooting ray that is closest to the root of

the decomposition tree. Descend the augmented factor graph as follows: at each node visited check either its $L(v)$ or $R(v)$ list; for each w in $L(v)$, test if the ray from q avoids the hourglass corresponding to the edge (v, w); at a leaf, no such hourglass exists but the edge of the splinegon intersected by the ray can be computed in $O(1)$ time. At first glance, it seems we need $O(\log^2 n)$ time. To achieve the $O(\log n)$ complexity, transform the factor graph so that it has bounded degree. Then, using fractional cascading, the $O(\log n)$ intersection tests between convex chains and the line can all be accomplished in $O(\log n)$ time. This algorithm differs from the original polygon algorithm [13] in only one respect. In the polygon algorithm, the test of whether a line intersects an hourglass is transformed to the dual problem of point inclusion in a convex polygon, solvable using a variant of binary search. Since no duality transforms are known to apply to curved objects, we solve the line-hourglass intersection problem directly using binary search on the two convex chains bounding the hourglass. □

THEOREM 2.6. *Given two query points p and q and the bounded degree decomposition, the factor graph, and the layered dag of a simple splinegon S and its pockets, the shortest path from p to q and its length can be reported in $O(\log n + k)$, where k is the number of segments in the path.*

Proof. The polygon algorithm of [25] extends directly. □

3. Separators. In this section we solve two optimum polygon separator problems and then generalize those solutions to accommodate splinegons. It should be clear that an area separator does not always exist. For example, there are polygons (splinegons) that cannot be bisected by a single segment. Our algorithm finds an optimum separator if one exists or reports the nonexistence otherwise. Given that we solve both problems in a uniform way, we describe the solution to the minimum length separator problem in depth and then refer briefly to the minimum sum of ratios problem.

If x and y delimit a separator for simple polygon (splinegon) P, then x and y are visible in P. Thus, if x lies on edge i and y on edge j, then i and j are visible in P. Furthermore, the line segment xy lies in the hourglass $H_{i,j}$ defined by the shortest paths from p_{j+1} to p_i ($SP_{p_{j+1}}(p_i)$) and from p_{i+1} to p_j ($SP_{p_{i+1}}(p_j)$). Thus our goal is to reduce the optimum separator problem for a simple polygon to a series of optimum separator problems on hourglasses that are simpler to solve.

Define as a_L (respectively, a_R) the area of P to the left (respectively, right) of the directed segment xy (see Fig. 9). Not every hourglass will admit a separator satisfying the constraints $a_L = K_L$ and $a_R = K_R$ for constants K_L and K_R, where $K_L + K_R = A$, and A is the area of P. Note, however, that $SP_{p_i}(p_{j+1})$ cuts the polygon into two or more pieces, some to its left and some to its right. The area of P to the left (respectively, right) of $SP_{p_{j+1}}(p_i)$ is denoted $AL_{p_{j+1}}(p_i)$ (respectively, $AR_{p_{j+1}}(p_i)$). Therefore a necessary and sufficient condition for H_{ij} to contain a separator xy is $AL_{p_{j+1}}(p_i) \leq K_L$, and $AR_{p_j}(p_{i+1}) \leq K_R$. Thus when we consider hourglass H_{ij}, we need to know both $AL_{p_{j+1}}(p_i)$ and $AR_{p_j}(p_{i+1})$. It is clear that computing these quantities for a specific H_{ij} could use $\Theta(n)$ time, leading to a time complexity of $O(n^3)$ for all $O(n^2)$ hourglasses. For a fixed vertex v, however, the shortest path map from v divides P into a linear number of triangles where the funnel $F_v(i)$ on edge i is the disjoint union of some subset of these triangles. Therefore, we can compute all $AL_v(p_i)$ for $i = 1, \cdots n$ incrementally in $O(n)$ time.

Below, we present a high-level description of our algorithm for computing the minimum-length separator, where $shortestpath(v)$ represents the procedure that computes the shortest path tree (map) from vertex v, $a(F)$ denotes the area of funnel F, and $hourglass(i, j, locmin)$ computes the minimum-length separator for $H_{i,j}$.

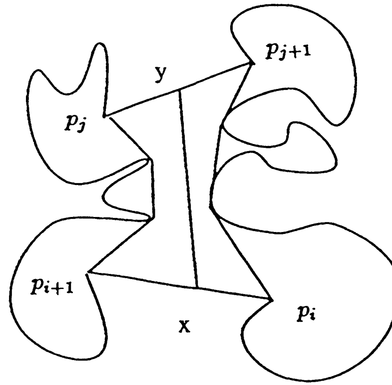


FIG. 9. *A polygon P and an area-separator xy.*

For $j = 1$ to n do begin
    $globmin = \infty$;
    $shortestpath(p_j)$;
    $shortestpath(p_{j+1})$;
    for $i = 1$ to n do
        if i, j are visible and $AL_{p_{j+1}}(p_i) \leq K_L$ and $AR_{p_j}(p_{i+1}) \leq K_R$
        then do begin
            $hourglass(i, j, locmin)$
            $globmin = MIN(globmin, locmin)$;
        end;
end;

Next, we need a procedure $hourglass(i, j, locmin)$ to solve the following problem:

> **Problem.** Given an hourglass $H_{i,j}$, find x and y on i and j, respectively, such that: (a) $xy \subseteq H_{i,j}$; (b) the area bounded by $SP_{p_{j+1}}(p_i)$, p_ix, xy, and yp_{j+1} equals $K_L - AL_{p_{j+1}}(p_i)$; and (c) the length of xy is minimum.

First, we simplify the test of condition (b). For each hourglass $H_{i,j}$, define $C_{p_{j+1}}(p_i)$ as the area of the region bounded by $SP_{p_{j+1}}(p_i)$ and the segment p_ip_{j+1}. Depth-first-search traversal of the shortest path tree of P from vertex p_{j+1} produces all of the $C_{p_{j+1}}(p_i)$ in linear time:

procedure $convex - area(v, s)$;
Begin
for all neighbors w of v do
    if the path from s to w is a counterclockwise convex chain
    then begin
        $C_s(w) := C_s(v) + area(\Delta svw)$;
        $convex - area(s, w)$;
    end;
End;

The test of condition (b) now reduces to determining whether the quadrilateral p_ixyp_{j+1} has area $K_L - AL_{p_{j+1}}(p_i) + C_{p_{j+1}}(p_i)$, from now on referred to as K.

Condition (a) is satisfied if and only if the closed halfplane to the left of the $\overrightarrow{xy}$ contains all the vertices of $SP_{p_{j+1}}(p_i)$ and the one to the right contains all the vertices of $SP_{p_{i+1}}(p_j)$. This condition could produce a linear number of constraints. To decompose the problem further, and thus reduce the number of constraints, we exploit the fact that shortest paths are convex. First, trim edge i to create an edge i' so that all of i' is visible from j. Next, subdivide i' into elementary segments by merging $S_{p_j}(i)$ and $S_{p_{j+1}}(i)$. As x moves from p'_i to p'_{i+1}, $anchor^{p_{j+1}}(x)$ and/or $anchor^{p_j}(x)$ changes only when x moves from one subinterval to the next. Thus, it is reasonable to refer to $anchor^{p_j}(I)$ and $anchor^{p_{j+1}}(I)$. Consequently, we can reduce an arbitrary hourglass problem to a series of problems defined on elementary hourglasses:

> **Problem.** For an elementary segment I of i, find points $x = (x_1, y_1)$ and $y = (x_2, y_2)$ on I and j, respectively, such that (a) the $anchor^{p_j}(I)$ and $anchor^{p_{j+1}}(I)$ do not lie in the same open halfspace defined by $\overrightarrow{xy}$, (b) the area of the quadrilateral $p_i x y p_{j+1}$ equals K, and (c) length of xy is minimum.

This is a continuous optimization problem, rather than a combinatorial one, which generates the following constraints, where $p_i = (k_1, l_1)$, $p_{j+1} = (k_2, l_2)$ and (a_i, b_i) represents the slope and y-intercept of the line containing edge i:

(1) $area(p_i x y p_{j+1}) = x_1 y_2 - x_2 y_1 + x_2 l_2 - k_2 y_2 + k_2 l_1 - k_1 l_2 + k_1 y_1 - x_1 l_1 = K$,

(2) x lies on line containing i: $y_1 = a_i x_1 + b_i$,

(3) y lies on line containing j: $y_2 = a_j x_2 + b_j$.

Substitution of (2) and (3) in (1) produces constants B_i such that

$$x_2 = \frac{B_1 x_1 + B_2}{B_3 x_1 + B_4}. \tag{4}$$

Substitution of (2), (3), and (4) in the expression for the length of xy,

$$L = \sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)},$$

gives the length as the square root of a rational function of one variable x_1. The domain of x_1 is restricted by the constraints that x must lie within the elementary segment I, the point y lies on edge j, and the line through x and y must keep the $anchor^{p_j}(I)$ and $anchor^{p_{j+1}}(I)$ on opposite sides. The length function L is the square root of a rational function of x_1. The degrees of the numerator and denominator of the rational function permit analytical solution for finding the optimum in constant time, even in the restricted domain. The global minimum for the original hourglass is the minimum of all the minima obtained from the continuous problems on elementary hourglasses. Therefore, the following theorem results.

THEOREM 3.1. *For an hourglass H_{ij} the minimum length separator problem can be solved in $O(h_{ij})$ time and space, where h_{ij} is the number of vertices of H_{ij}.*

Since the size of an hourglass H_{ij} is in the worst case $O(n)$, where n is the number of vertices of polygon P, and since we call the hourglass algorithm at most $O(n^2)$ times, the time complexity of the entire separator algorithm is at most $O(n^3)$. We exploit the linearity of the shortest path trees to improve that bound.

THEOREM 3.2. *The minimum length area separator of a simple polygon can be computed in $O(n^2)$ time and $O(n)$ space.*

Proof. We have $O(n)$ calls to shortest path algorithm for a total of $O(n^2)$ time. $O(n^2)$ time is spent computing the $AL_v(w)$. Finally, for a particular hourglass H_{ij}, we spend $O(s_j^{p_i} + s_j^{p_{i+1}})$. Thus the whole algorithm takes

$O(\sum_{i=1}^{n}\sum_{j=1}^{n}(s_j^{p_i} + s_j^{p_{i+1}}))$ which is clearly $O(n^2)$, since

$$\sum_{k=1}^{n} s_k^v = O(n)$$

for any vertex v. □

An alternate proof of the theorem uses the following lemma, which is interesting in its own right:

LEMMA 3.3. *The sum of the sizes of all (open) hourglasses of a simple polygon P is $O(n^2)$.*

Proof. Let h_{ij} be the size of the hourglass defined by the visible edges i and j and let *ad*, *bc* the inner common tangent segments of the convex chains of the hourglass (see Fig. 10). But

$$s_j^{p_i} = ab + p_{j+1}b + p_j d, \tag{1}$$

$$s_i^{p_j} = cd + p_{i+1}c + p_i a. \tag{2}$$

(1) and (2) imply $s_j^{p_i} + s_i^{p_j} = h_{ij}$.
Then

$$\sum_{i=1}^{n}\sum_{j=1}^{n} h_{ij} = \sum_{i=1}^{n}\sum_{j=1}^{n}(s_j^{p_i} + s_i^{p_j}).$$

The last sum is $O(n^2)$, since

$$\sum_{k=1}^{n} s_k^v = O(n)$$

for any vertex v. □

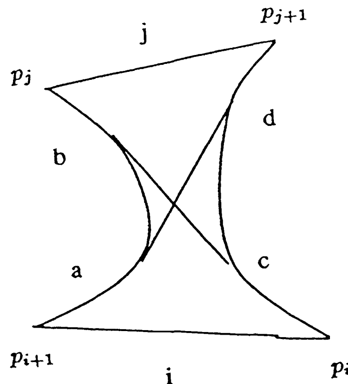


FIG. 10. *Hourglass between edges i and j and the inner common tangents ad and bc.*

The alternate algorithm works as follows. Find the shortest path maps from p_j and p_{j+1}, respectively. Merge $S_{p_j}(i)$ and $S_{p_{j+1}}(i)$ for every i into a linear-sized subdivision M. If x moves along an elementary segment I of M, the anchors of x with respect to the endpoints j remain unchanged. Thus we can unambiguously refer to $anchor^{p_j}(I)$ and

$anchor^{p_{j+1}}(I)$. For each I of M, perform the following simple test: If $anchor^{p_{j+1}}(I) <> anchor^{p_j}(I)$, then for every point x on I, x is visible from edge j; for each segment I solve the continuous optimization problem discussed above. Repeat the same process for all edges j.

We now generalize this result to splinegons.

THEOREM 3.4. *The minimum length area separator of a simple splinegon can be computed in $O(n^2)$ time and $O(n)$ space.*

Proof. The hourglass problem for splinegons becomes somewhat more difficult, since the anchor can be a curved segment, rather than a single vertex. Nonetheless, it is possible to alter the constraints of the elementary hourglass problem to require that a curved segment lie above a line rather than that a point lie above a line. The revised problem can still be solved in constant time. For a particular fixed edge i of P we have to scan all elementary segments I of the merged subdivision M. For each such segment we have to solve a continuous optimization problem which takes $O(1)$ time to solve. Because of the linearity of the shortest path trees, we have $O(n)$ elementary segments, which implies that it takes $O(n)$ time to find a separator xy when y lies on a fixed edge i. Summing all over edges i we get an $O(n^2)$ time algorithm. □

THEOREM 3.5. *The minimum sum of ratios separator of a simple polygon P of n vertices can be computed in $O(n^2)$ time and $O(n)$ space.*

Proof. The minimum sum of ratios separator problem can be solved in a fashion similar to that used to solve the minimum-length separator problem. A new, easily computed parameter is needed: $LP_{p_{j+1}}(p_i)$ represents the length of the boundary of P from p_{j+1} to p_i in clockwise order. All of the constraints in the continuous optimization problem remain unchanged, but we need to minimize a more complicated expression. The area a_L to the left of xy equals area$(p_i x y p_{j+1}) + AL_{p_{j+1}}(p_i) - C_{p_{j+1}}(p_i)$. The perimeter p_L to the left of xy is $LP_{p_{j+1}}(p_i) + length(p_i x) + length(y p_{j+1}) - length($all polygon edges on $xy)$. Compute a_R and p_R comparably. We need to minimize $(a_L/p_L^2) + (a_R/p_R^2)$. The expression can be reduced to an optimization problem in two variables which can be solved with classical methods. □

We can combine our method with Hershberger's method for finding the visibility graph of a simple polygon to improve our result to $O(m)$ time, where m is the size of the visibility graph of polygon P. In order to do that, we fix a particular edge j and find the shortest path map from one of its endpoints, say, p_j. Then starting from the other endpoint p_{j+1}, we construct the shortest path map from p_{j+1} incrementally, as in [29]. In order to combine it with our method, at each step, update the appropriate areas and solve the corresponding continuous optimization problem. Repeat the whole process for every j.

THEOREM 3.6. *The minimum length area separator for a simple polygon can be solved in $O(m)$ time and $O(n)$ space, where m is the size of the visibility graph of the polygon.*

Using the same reasoning we can prove the following.

THEOREM 3.7. *The minimum sum of ratios separator of a simple polygon P of n vertices can be computed in $O(m)$ time and $O(n)$ space.*

4. Inscribed triangles. For three points x, y, z on the boundary of a simple polygon P to define an inscribed triangle, it is necessary and sufficient that they be pairwise visible. If x, y, z lie on edges k, i, j, respectively, then the points are pairwise visible if and only if xy, yz, and zx lie inside $H_{k,i}$, $H_{j,i}$, and $H_{k,j}$, respectively. Thus the boundary of the triangle xyz is interior, both to P and to the union of the three hourglasses. Since P is simple, the entire triangle must be interior to P. It is also contained in the polygon $F_{i,j,k} \subseteq P$ bounded by i, $SP_{p_{i+1}}(p_k)$, k, $SP_{p_{k+1}}(p_j)$, j and $SP_{p_{j+1}}(p_i)$.

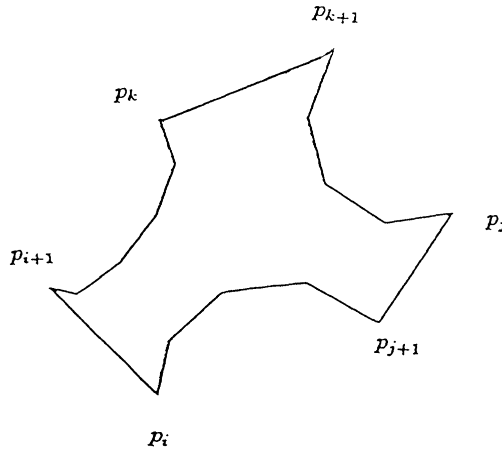


FIG. 11. *A fan-shaped polygon F inside a simple polygon P.*

$F_{i,j,k}$ is called a *fan-shaped polygon* with *bases* i, j, k (see Fig. 11). $F_{i,j,k}$ is *legal* if and only if the visible parts of each of its bases with respect to the two others have nonempty intersections. A triangle T is *inscribed* in $F_{i,j,k}$ if and only if $T \subseteq F_{i,j,k}$ and the vertices of T lie on the bases. Every triangle inscribed in a simple polygon P is also inscribed in a fan-shaped polygon $F_{i,j,k} \subseteq P$. Below we present a high-level description of our algorithm for computing the maximum area/perimeter inscribed triangle, where $shortestpath(v)$ represents the procedure that computes the shortest path tree (map) from vertex v, and $fan(i, j, k, locmax)$ computes the maximum triangle inscribed in a legal $F_{i,j,k}$.

```
globmax = 0;
for i = 1 to n do begin
    shortestpath(p_i); shortestpath(p_{i+1});
    for j = 1 to n do begin
        shortestpath(p_j); shortestpath(p_{j+1});
        for k = 1 to n do begin
            if F_{i,j,k} is legal (if edges i, j, k are pairwise visible)
            then fan(i,j,k, locmax);
            globmax = MAX(globmax, locmax);
        end;
    end;
end;
```

It remains to develop the procedure $fan(i, j, k, locmax)$.

LEMMA 4.1. *For a fan-shaped polygon $F_{i,j,k}$ the maximum-area inscribed triangle with vertices x, y, z on the bases, must have at least two sides tangent to the convex chains of the fan.*

Proof. By contradiction. Specifically, assume that neither xy nor xz is tangent to the boundary of $F_{i,j,k}$. Tangents from y and z to the chains $SP_{p_i}(p_{k+1})$ and $SP_{p_{j+1}}(p_k)$, respectively, intersect k at points v and w such that x must lie between them. Then it is clear that either vyz or wyz must have area greater than or equal to the area of xyz (the equality happens when k is parallel to yz) (see Fig. 12). ☐

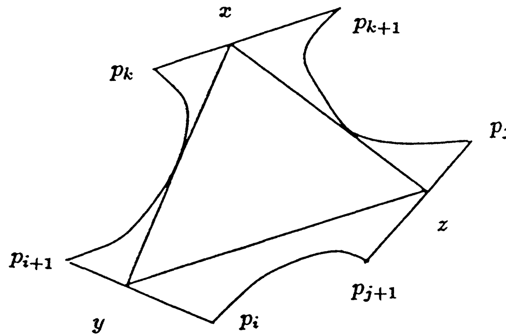


FIG. 12. *The maximum triangle has two sides tangent to the inward convex chains.*

LEMMA 4.2. *Assume we are given two line segments AB and CD. Let x be a point on CD. Then the triangle xAB has maximum perimeter L when either $x = C$ or when $x = D$.*

Proof. The locus of points x such that triangle xAB has perimeter L is an ellipse with foci A and B which contains CD but has at least one point of CD on its boundary. Either C or D lies on the boundary. □

LEMMA 4.3. *For a fan-shaped polygon, the maximum-perimeter inscribed triangle with vertices on the bases must have at least two sides tangent to the convex chains of the fan.*

Proof. Bring tangents as we did in Lemma 4.1 and then apply Lemma 4.2. □

Assume, without loss of generality, that the optimum triangle Δxyz has xy tangent to $SP_{p_{i+1}}(p_k)$, and xz tangent to $SP_{p_j}(p_{k+1})$. To find the optimum, trim edge k to create a maximal edge k', every point of which is visible from both i and j. If k' is empty, then no inscribed triangle exists. Perform comparable operations on edges i and j, creating i' and j', respectively. Next, subdivide k' by merging $S_{p_{j+1}}(k)$, $S_{p_j}(k)$, $S_{p_i}(k)$, $S_{p_i}(k)$. Call the resulting subdivision M. As x moves from p'_k to p'_{k+1}, $anchor^{p_{i+1}}(x)$ and/or $anchor^{p_j}(x)$ changes only when x moves from one subinterval to the next. Consequently, we can reduce an arbitrary fan-shaped polygon problem to a series of problems defined on simpler fan-shaped polygons.

> **Problem.** For each interval K of k', find points $x \in K$, $y \in i$, $z \in j$ such that (a) $anchor^{p_{i+1}}(K) \in xy$ and $anchor^{p_j}(K) \in xz$, (b) y and z are mutually visible, and (c) area/perimeter of xyz is maximum.

To test condition (b), we must be able to detect possible intersections of yz with the boundary of P, in particular with the convex chain $SP_{p_{i+1}}(p_j)$.[1] This process could still be difficult, so we choose to decompose the problem further. Subdivide i' according to $S_{p_j}(i)$ and subdivide j' according to $S_{p_{i+1}}(j)$. As y (respectively, z) moves from p'_i to p'_{i+1} (respectively, from p'_j to p'_{j+1}) $anchor^{p_{j+1}}(y)$ (respectively, $anchor^{p_i}(z)$) changes only when y (respectively, z) moves from one subinterval to the next. Let W be the number of vertices of either $S_{p_{j+1}}(p_i)$ or $S_{p_i}(p_{j+1})$. To each interval of i' (respectively, j') assign a number called its rank, which corresponds to the position of its anchor in $SP_{p_i}(p_{j+1})$ (respectively, $SP_{p_{j+1}}(p_i)$), assuming that the first position is 0.

[1]The segment yz cannot intersect chains $SP_{p_i}(p_k)$ and $SP_{p_{j+1}}(p_k)$, since both chains lie outside of the convex angle yxz.

Refine the subdivision of k' further so that whenever $x \in K$, y and z each have constant rank. Let the rank of K equal the sum of those ranks. The algorithm is straightforward. If $rank(K) < W$, then do nothing. Since y, z are not visible from each other. If $rank(K) > W$, solve Problem I; if $rank(K) = W$ solve Problem II.

Problem I. Given three nonintersecting line segments AB, CD, and EF and two points p, q such that p (respectively, q) lies on AE and BF (respectively, AC and BD), find *s*,*t*,*u* on AB,EF,CD with p (respectively, q) on st (respectively, su), such that the area of stu is maximum.

Problem II. Add the constraint that the line through t and u should always be above a constant point (x_0, y_0).

Solutions to both problems can be computed analytically.

Solution of Problem I. The objective is to compute the coordinates of s, t and u. Call the coordinates of these points x_1, y_1, x_2, y_2, and x_3, y_3, respectively. Let (k_1, l_1) (respectively, (k_2, l_2)) be the coordinates of point p (respectively, q).

Points s, t, u lie on three different lines:

$$y_1 = a_1 x_1 + b_1, \quad (1)$$
$$y_2 = a_2 x_2 + b_2, \quad (2)$$
$$y_3 = a_3 x_3 + b_3. \quad (3)$$

Segments st and su pass through points p and q, respectively:

$$x_1 y_2 - x_2 y_1 - k_1(y_2 - y_1) - l_1(x_1 - x_2) = 0, \quad (4)$$
$$x_1 y_3 - x_3 y_1 - k_2(y_3 - y_1) - l_2(x_1 - x_3) = 0. \quad (5)$$

Algebraic manipulation of equations 2, 3, 3, 5, 5 produces constants A_i and B_i such that

$$x_2 = \frac{A_4 - A_2 x_1}{A_1 x_1 - A_3}, \quad (6)$$
$$x_3 = \frac{B_4 - B_2 x_1}{B_1 x_1 - B_3}. \quad (7)$$

The area of a triangle is given by

$$A = x_3 y_2 - x_2 y_3 + x_1 y_3 - x_3 y_1 + x_2 y_1 - x_1 y_2. \quad (8)$$

Substitution of the above equations into the last one produces a rational function of one variable x_1 where the numerator is a polynomial of degree three and the denominator is of degree two. The extrema can be computed using calculus: the derivative of this function is a rational function where the numerator is of degree at most four, but the roots of any algebraic equation on one variable up to degree four can be found in closed form. The domain of variable x_1 is fixed by the facts (a) that s, t, u lie on AB, EF, CD, respectively, and (b) that points k and l are always on sides st and su, respectively. □

Solution to Problem II. In addition to the equations above, we have the new constraint

$$y_0 \leq \frac{y_2 - y_1}{x_2 - x_1} x_0 + \frac{x_1 y_2 - x_2 y_1}{x_2 - x_1}. \quad (9)$$

If we express everything in terms of x_1 as we did before we get

$$(a_2 - a_1)x_1 \frac{A_4 - A_2x_1}{A_1x_1 - A_3} + (b_2 - a_1x_0 + y_0)x_1 + (a_2x_0 - b_1 - y_0)\frac{A_4 - A_2x_1}{A_1x_1 - A_3} - b_1x_0 \geq 0, \tag{10}$$

which is a rational function with the numerator a polynomial of degree two and the denominator a polynomial of degree one. Therefore the zeros and the intervals of interest can be computed analytically. □

The perimeter optimization problem also generates two types of subproblem with these same constraints, but a different, and more complicated, objective function. As rewritten as a function of x_1, the roots of the first derivative cannot be found analytically; one of the classical methods for root finding from numerical analysis must be used. We assume that finding the roots of an equation by a numerical method takes $O(1)$ time.

We conclude that the continuous rotation of the triangle xyz produced as x is moving along the "legal" portion of AB can be discretized into a finite number of problems discussed below, each of which can be solved analytically in $O(1)$ time.

LEMMA 4.4. *The maximum inscribed triangle in a fan-shaped polygon can be found in* $O(s_k^{p_i} + s_k^{p_{j+1}} + s_j^{p_{i+1}} + s_i^{p_j})$*, which is* $O(n)$*, where* n *is the number of vertices of the fan-shaped polygon.*

Since we decompose the simple polygon into at most $O(n^3)$ fan-shaped polygons, computing the maximum inscribed triangle in the simple polygon uses at most $O(n^4)$ time. Careful analysis produces a better bound.

THEOREM 4.5. *The maximum triangle inscribed in a simple polygon* P *can be found in* $O(n^3)$ *arithmetic operations, where* n *is the number of vertices of* P*. The space required is* $O(n)$*.*

Proof. The total time spent in the shortest path computation is $O(n^3)$, since the shortest path procedure is called $O(n^2)$ times. Each **if** statement takes $O(1)$ time, since the "legality" of $F_{i,j,k}$ can be decided from the shortest path computation. According to Lemma 4.4, the procedure *fan(i,j,k)* takes $O(s_j^{p_i} + s_i^{p_{j+1}} + s_k^{p_{i+1}} + s_k^{p_j})$. Thus the total time spent on the fan-shaped polygons corresponding to all triples (i, j, k) of edges of P is:

$O(\sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n (s_j^{p_i} + s_i^{p_{j+1}} + s_k^{p_{i+1}} + s_k^{p_j}))$, which is $O(n^3)$, since $\sum_{k=1}^n s_k^v = O(n)$ for any vertex v of P [26]. □

An alternative way of proving the above bound is to prove the following combinatorial result of independent interest.

LEMMA 4.6. *The sum of the sizes of all legal fan-shaped polygons with three bases of a simple polygon* P *is* $O(n^3)$*.*

Proof. Let f_{ijk} be the size of a legal 3-fan-shaped polygon defined by the edges i, j, k of a simple polygon P. Consider all pairs of hourglasses defined by the edges i, j, k. These hourglasses can be formulated by considering the common tangents of the chains of the fan-shaped polygons. Let h_{ij}, h_{kj}, h_{ik} be the sizes of the corresponding hourglasses. For our proof, $C_{p_ip_{j+1}}$ will represent both the chain $C_{p_ip_{j+1}}$ and its length. Consequently,

(1) $h_{ij} = p_{i+1}a + 1 + p_jb + C_{p_ip_{j+1}}$,
(2) $h_{ik} = p_{k+1}d + 1 + p_ic + C_{p_kp_{i+1}}$,
(3) $h_{kj} = p_{j+1}f + 1 + p_ke + C_{p_jp_{k+1}}$.

Summing up (1), (2), and (3) produces

$h_{ij} + h_{ik} + h_{kj} = 3 + p_{i+1}a + p_jb + p_{k+1}d + p_ic + p_{j+1}f + p_ke + f_{ijk}$,

which implies

(4) $f_{ijk} < h_{ij} + h_{ik} + h_{kj}$.

Then

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} f_{ijk} < \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}(h_{ij} + h_{ik} + h_{kj}).$$

This last summation is $O(n^3)$, since according to Lemma 3.3,

$$\sum_{i=1}^{n}\sum_{j=1}^{n} h_{ij} = O(n^2). \qquad \square$$

LEMMA 4.7. *The sum of the sizes of all legal fan-shaped polygons with k bases of a simple polygon P is $O(n^k)$.*

Proof. By induction on k. $\square$

The complexity of our method depends on two components: (a) shortest path computation and (b) computation done in all fan-shaped polygons. At all times, either we have computed the visible part of P from both edges i and j, or we are in the process of computing it. In each of (a) and (b) we expend $O(n^3)$ time. Now we are going to see how the Hershberger technique [29] of creating shortest paths incrementally can profitably be combined with our method of the previous section.

First, we define some terms and notation. Let i and j be two polygon edges. The number of edges between i and j as we move clockwise from i to j is called the *distance* between i and j. Let D^i denote the symmetric difference of the shortest path maps from the endpoints of edge i. Also let d_j^i be the part of D^i that intersects edge j.

Assuming that the shortest path map from vertex p_1 has already been computed [26], Hershberger [29] shows that the shortest path map from p_2 can be computed incrementally by scanning the boundary of P in counterclockwise order, starting from p_2 and going back to p_1, in time proportional to the symmetric difference of the two shortest path maps D^1. Consequently, he proves that the total number of differences of all shortest path maps $\sum_{i=1}^{n} |D^i|$ is $O(m)$, where m is the size of the visibility graph of P. Additionally, at any moment his algorithm satisfies the following invariant: upon reaching a particular point x on the boundary of P, the shortest path map from p_1 within the area of P to the left of $SP_{p_1}(x)$ is known, and the shortest path map from p_2 within the area of P to the right of $SP_{p_2}(x)$ is known.

We use these results as follows. Assume that edges i and j have distance d. Assume also that we are given the shortest path maps from p_{j+1} and p_{i+1}. Move from p_j to p_i clockwise, using Hershberger's method to construct part of the shortest path map from p_j using the shortest path map from p_{j+1}. Then start scanning the boundary of P from p_i clockwise up to p_{j+1}. During this walk, we concurrently begin constructing the shortest path map from p_i using the one from p_{i+1} and finishing the one from p_j using the one from p_{j+1}. When we walk on edge k, we want to apply our method of the previous section on fan-shaped polygon f_{ijk} (see Fig. 13). In order to do that, we need to know the elementary intervals for the continuous optimization problems, i.e., we need to know $S_{p_{j+1}}(k)$, $S_{p_j}(k)$, $S_{p_i}(k)$, $S_{p_{i+1}}(k)$. But the walk we described above will have produced these subdivisions. Specifically $S_{p_j}(k)$ can be derived from $S_{p_{j+1}}(k)$ and $S_{p_i}(k)$ from $S_{p_{i+1}}(k)$. Furthermore, we need to check the visibility of $y \in i$ and $z \in j$ using $S_{p_j}(i)$ and $S_{p_{i+1}}(j)$. But since we are on edge k, which is "after" edge i and "before" edge j, both $S_{p_j}(i)$ and $S_{p_{i+1}}(j)$ are known. After a complete cycle on the boundary, we have constructed the shortest path maps from p_i and p_j. Then we

can proceed by advancing both edges i and j counterclockwise and repeating the same procedure. The above procedure suggests the following lemma.

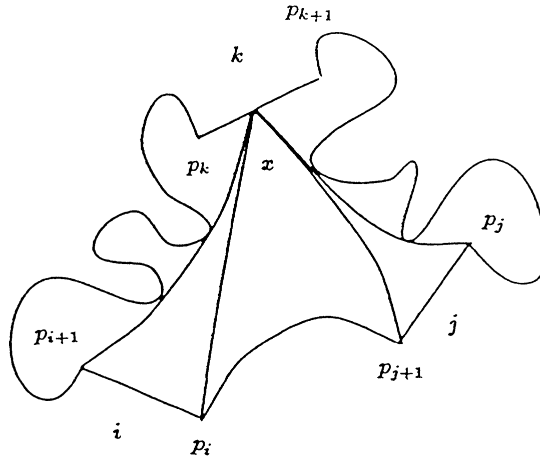


FIG. 13. *Modified fan-shaped algorithm.*

LEMMA 4.8. *The time spent in shortest path computations for fan-shaped polygons* f_{ijk} *when edges* i *and* j *have fixed distance is* $O(n+m)$.

Repeating the above algorithm for pairs i and j of all possible distances $1, \cdots, n$ yields:

LEMMA 4.9. *The time spent for all shortest path computations is* $O(n^2+nm)$.

The question that remains is whether it is possible to reduce the total cost of the fan-shaped computations, i.e., the total number of continuous optimization problems we have to solve.

According to Lemma 4.4 we know that the time spent in a fan-shaped polygon f_{ijk} is $O(s_k^{p_i} + s_k^{p_{j+1}} + s_j^{p_{i+1}} + s_i^{p_j})$ and all these quantities sum up to $O(n^3)$, according to Theorem 4.5. But $|D^i| = \sum_{j=1}^{n} d_j^i$, and $\sum_{i=1}^{n} |D^i| = O(m)$.

LEMMA 4.10. *The maximum inscribed triangle in a fan-shaped polygon can be found in* $O(d_k^i + d_k^j + d_j^i + d_i^j)$.

LEMMA 4.11. *The total time spent in all fan-shaped polygons of a polygon* P *is* $O(nm)$, *where* n, m *is the number of vertices and the size of visibility graph of* P, *respectively.*

Proof. According to Lemma 4.10, the time spent in a fan-shaped polygon f_{ijk} is $O(d_k^i + d_k^j + d_j^i + d_i^j)$. Summing over all fan-shaped polygons,we get

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} (d_k^i + d_k^j + d_j^i + d_i^j).$$

Since $\sum_{i=1}^{n} \sum_{j=1}^{n} d_j^i = O(m)$, the above summation is $O(mn)$. □

THEOREM 4.12. *The maximum triangle inscribed in a simple polygon* P *can be found in* $O(n^2+nm)$ *arithmetic operations where* n *is the number of vertices of* P *and* m *is the size of the visibility graph. The space required is* $O(n)$.

Unfortunately, the maximum area or perimeter triangle inscribed in a simple splinegon might not have two sides tangent to the chains of the fan-shaped splinegon, as was true in the polygon case. In fact, we present a construction where none of the sides of any maximum inscribed triangle is tangent to the fan-shaped polygon.

Construction. Consider two circles C_1 and C_2 with common center O and radii R_1 and R_2, respectively, such that $R_1 < R_2$. Let xyz be an equilateral triangle inscribed in C_2. Let A, B, C, D, E, F be the intersection points of C_1 with the sides of $\triangle xyz$ as shown in Fig. 15. Let A', B' be points on C_1 counterclockwise and clockwise, respectively, from A, B. Similarly define points C', D', E', F', see Fig. 15. Define a convex curve segment with one endpoint A' the other endpoint B' so that it lies entirely between C_1 and C_2 and intersects C_2 at a unique point x. Define similar curved segments with endpoints C', D' and E', F'. Define a concave curve segment between each pair of points (A', F'), (B', C'), (D', E') so that they do not intersect $\triangle xyz$. The above six curved segments (three convex and three concave) define a splinegon S that lies entirely inside circle C_2 except for points x, y, z, which lie on the boundary of C_2.

Claim. $\triangle xyz$ is the maximum area triangle inscribed into S.

Proof. By contradiction. Assume there exists a $\triangle abc$ such that $area(\triangle abc) > area(\triangle xyz)$. Then, at least one of the vertices a, b, c lies in the interior of C_2. But then there exists at least one triangle T inscribed in C_2 such that $area(\triangle abc) < area(\triangle T)$. It is well known, however, that a maximum area triangle inscribed in a circle is equilateral. Thus $area(\triangle T) \leq area(\triangle xyz)$, which implies that $area(\triangle abc) < area(\triangle xyz)$. □

In the polygon case, it was possible to reduce the number of triples of elementary segments considered within a single fan-shaped polygon to linear in the size of that polygon. In the splinegon case, all $O(n^3)$ triples of elementary segments must be considered.

THEOREM 4.13. *The maximum area or perimeter triangle inscribed in a simple splinegon can be found in $O(n^4)$ time and $O(n)$ space.*

Proof. Move point x along the legal part of k. Each interval of this subdivision corresponds to specific anchors (although these anchors may be curved segments rather than points) of the shortest paths from p_i, p_{i+1}, p_j, p_{j+1} to x. The line through xy (xz) must have the anchors of x with respect to p_i and p_{i+1} (respectively, p_j and p_{j+1}) in opposite sides in order to guarantee visibility of x, y (respectively, x, z). To check the visibility of y and z, consider all pairs of segments on the subdivision of p_ip_{i+1} and p_jp_{j+1}.

Each fan-shaped splinegon requires $O((s_k^{p_{i+1}} + s_k^{p_j} + s_k^{p_i} + s_k^{p_{j+1}})s_i^{p_{j+1}}s_j^{p_i})$ time. Since $s_k^{p_i} \leq n$, the total complexity is

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}((s_k^{p_{i+1}} + s_k^{p_j} + s_k^{p_i} + s_k^{p_{j+1}})s_i^{p_{j+1}}s_j^{p_i}).$$

Since $s_k^{p_i} < n$, the above sum is $O(n^4)$. □

This result is significant in that it is the first instance in which there is an apparent asymptotic gap between polygon and splinegon solutions for the same problem.

We now move to the maximum constrained inscribed triangle problem.

LEMMA 4.14. *The Maximum Inscribed Triangle with one of its sides having given length has at least one of the nongiven length sides tangent to a fan-shaped polygon.*

Proof. Similar to that of Lemmas 4.1 and 4.3. □

THEOREM 4.15. *For a simple polygon P of n vertices, the Maximum Inscribed Triangle with one of its sides having given length can be found in $O(n^3)$ time and $O(n)$ space.*

Proof. The algorithm for this problem uses techniques similar to those used to solve the unconstrained problems. As in the case of maximum area inscribed triangle, edge k is subdivided by the shortest path maps from the source points $p_{j+1}, p_j, p_{i+1}, p_i$ into elementary intervals such that for every point x in an elementary interval K, $anchor^{p_{j+1}}(x)$, $anchor^{p_j}(x)$, $anchor^{p_{i+1}}(x)$, $anchor^{p_i}(x)$ remain unchanged. As we did before instead

of $anchor^{p_i}(x)$ we will refer to $anchor^{p_i}(K)$. Assume that triangle vertices x, y, z lie on edges k, i, j, respectively, and that yz is a non-fixed-length triangle side that remains tangent to $SP_{p_{j+1}}(p_i)$ (see Fig. 14). The initial position of y coincides with the closest point to p_i which is visible from edge j. Rotate yz clockwise so that it remains always tangent to $SP_{p_{j+1}}(p_i)$. At any moment $anchor^{p_i}(z) = anchor^{p_{j+1}}(y)$. As in the case of edge k, both edges i and j are subdivided into elementary intervals of constant anchor. Thus let I (respectively, J) be the elementary intervals where y (respectively, z) belong. For every pair of elementary intervals I and J, consider all elementary intervals K of edge k. For each such triple (I, J, K), solve the following continuous constant size optimization problem in $O(1)$ time.

Problem. Given three line segments K, I, J and five fixed points A, B, C, D, E, compute the coordinates of points $x = (x_1, x_2)$ on K, $y = (y_1, y_2)$ on I, and $z = (z_1, z_2)$ on J such that: (a) xy has constant length L, (b) $\overrightarrow{xy}$ (respectively, $\overrightarrow{xz}$) keeps A and B (respectively, C and D) on opposite halfplanes, (c) $\overline{yz}$ contains E, and (d) the area of xyz is maximized. (A, B, C, D represent $anchor^{p_i}(K)$, $anchor^{p_{i+1}}(K)$, $anchor^{p_j}(K)$, $anchor^{p_{j+1}}(K)$, respectively, and E represents $anchor^{p_i}(J) = anchor^{p_{j+1}}(I)$.)

Solution. Since points x, y, z lie on given lines, x_2, y_2, z_2 are expressed in terms of x_1, y_1, z_1. Since yz passes through a given point E such that $E = anchor^{p_i}(J) = anchor^{p_{j+1}}(I)$, then z_1 is expressed in terms of y_1. Since xy has length L, y_1 can be expressed in terms of x_1. All of these substitutions produce a one-variable optimization problem. Finally, we add the visibility constraints for the x, y, z described in part (b) of the statement of the problem, generating four one-variable constraints.

Using the same notation as in the previous section, the number of elementary intervals K is $O((s_k^{p_{i+1}} + s_k^{p_i} + s_k^{p_j} + s_k^{p_{j+1}}))$ and the number of interval pairs (I, J) is $O(s_i^{p_{j+1}} + s_j^{p_i})$, making the complexity for the fan-shaped polygon $O((s_k^{p_{i+1}} + s_k^{p_i} + s_k^{p_j} + s_k^{p_{j+1}})(s_i^{p_{j+1}} + s_j^{p_i}))$. The complexity of the whole algorithm is

$$\sum_{i=1}^{n}(O(n) + \sum_{j=1}^{n}(O(n) + \sum_{k=1}^{n}(O((s_k^{p_{i+1}} + s_k^{p_i} + s_k^{p_j} + s_k^{p_{j+1}})(s_i^{p_{j+1}} + s_j^{p_i}))))) = O(n^3). \square$$

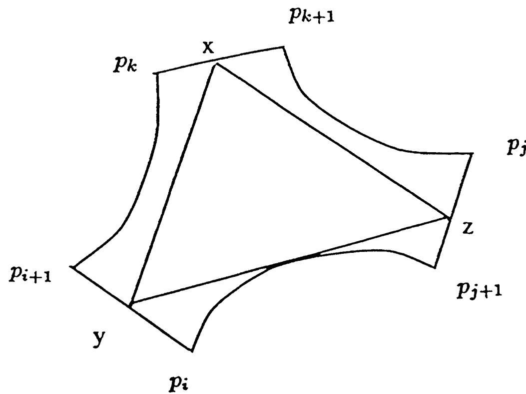


FIG. 14. *Triangle side xy is of constant length and yz remains tangent to the convex chain from p_{j+1} to p_i.*

Remark. For splinegons, as in the case of the maximum area inscribed triangle, we do not have the tangency property of at least one edge of the constrained inscribed triangle. It is not therefore hard to see that for a simple splinegon P of n vertices, the Maximum Inscribed Triangle with one of its sides having given length can be solved in $O(n^4)$ time and $O(n)$ space.

5. Minimum area concave quadrilateral. In this section, we present algorithms to compute the minimum-area *nondegenerate* concave quadrilateral circumscribing a simple polygon P, if one exists. $ABCD$ will always represent a concave quadrilateral where C is the reflex vertex. We always seek *nondegenerate* quadrilaterals such that A, B, C, D are distinct points and no three of them are collinear.[2] $CH(P)$ represents the convex hull of a simple polygon P. Each simple polygonal or splinegonal region Q interior to $CH(P)$ but exterior to P is called a *pocket* of P. Each edge of $CH(P)$ that is not an edge of P is called a *pseudoedge* of P. The following lemmas provide characterizations of the minimum-area concave quadrilateral.

LEMMA 5.1. *If $ABCD$ is a minimum-area concave quadrilateral containing a simple non-convex polygon or splinegon P, then A, B, D are not in the interior of $CH(P)$, with each of AB and AD is tangent to $CH(P)$ at points k and l, respectively. (AB (respectively, AD) may contain a whole edge of $CH(P)$, not just a single point k, (respectively, l)). (See Fig.* 16.)

Proof. By definition $CH(P)$ is the minimal convex set which contains P. That $\triangle ABD$ is a convex object containing P, implies that $CH(P) \subseteq \triangle ABD$. If either AB or AD is not tangent to P, then B can be moved, reducing the area of both ABD and $ABCD$. □

LEMMA 5.2. *The reflex vertex C, of a minimum-area quadrilateral containing a simple non-convex polygon P, lies inside the visibility polygon VQ_i of some pocket Q_i of P with respect to pseudoedge $v_i w_i$.*

Proof. It suffices to prove that C lies in some pocket Q_i of P. If that is true, then it is clear that C lies in the visibility polygon of Q_i with respect to pseudoedge $v_i w_i$, since both BC and DC intersect $v_i w_i$. Let $ABCD$ be a minimum nondegenerate quadrilateral which contains P and assume C is not inside $CH(P)$. (See Fig. 17.) Then at least one of BC or CD contains no point of $CH(P)$. Rotate that edge around C by a small angle θ so that B or D, respectively, is closer to A. The new quadrilateral is smaller than the original, producing a contradiction. □

LEMMA 5.3. *Sides BC, DC are tangent to the boundary of a pocket Q_i with pseudoedge $v_i w_i$ at points a, b distinct from C where $a = anchor^{v_i}(C)$ and $b = anchor^{w_i}(C)$.*

Proof. Assume that C lies inside some pocket Q_i of P but that BC and DC are not tangent to the pocket boundary. C is visible within Q_i from $v_i w_i$, which implies that the shortest paths from v_i to C and from w_i to C inside Q_i are inward convex chains. Let a and b be the anchors of these two shortest paths. Assume that BC and DC do not pass through a and b, respectively. Let B' (respectively, D') be the intersections of Ca and Cb with AB and AD, respectively. Since B' (respectively, D') is between A and B (respectively, A and D), the area of $AB'CD'$ is less than the area of $ABCD$. □

LEMMA 5.4. *If $ABCD$ is a minimum-area concave quadrilateral containing a simple polygon P, then the following hold:*[3]

(a) *The midpoint of AB (respectively, AD) lies on $CH(P)$;*
(b) *The midpoint of BC (respectively, DC) either lies on Q_i or it lies between two distinct points of tangency on BC relative to Q_i.*

Proof.

(a) According to Lemma 5.1, AB is tangent to $CH(P)$. Let m (respectively, n) be the common point of AB and $CH(P)$ closest to A (respectively, B). (Note that m and n may be identical.) Assume that the midpoint of AB does not lie

---

[2] We exclude the collinear case, since this reduces in finding the minimum area triangle containing a convex polygon, a problem solved in [37].

[3] The lemma does not necessarily hold for splinegons.

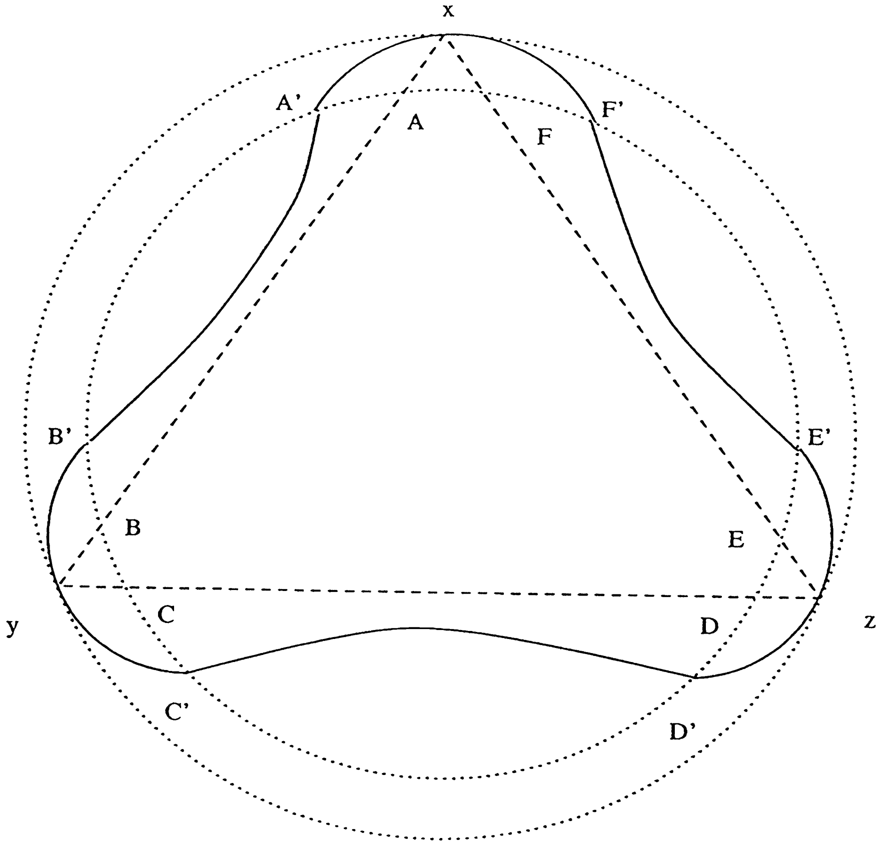


FIG. 15. *A maximum area or perimeter triangle inscribed in a simple splinegon might not have two sides tangent to the chains of the fan-shaped splinegon, as was true in the polygon case.*

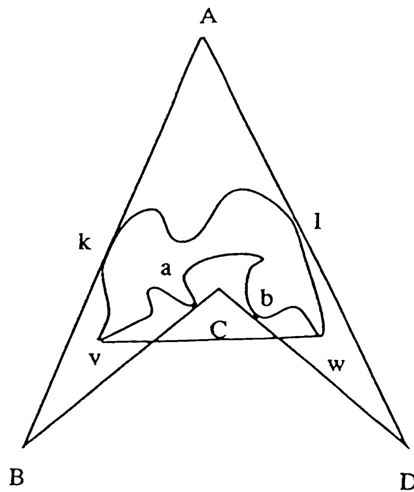


FIG. 16. *The minimum-area concave quadrilateral containing P.*

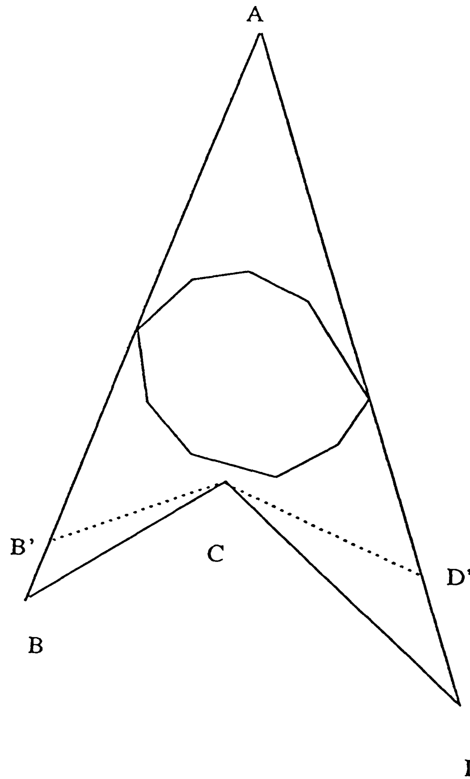


FIG. 17. *Reflex vertex C should lie in the interior of $CH(P)$.*

between m and n. Assume without loss of generality that $An < nB$. Then rotate AB counterclockwise around n by a very small angle θ. Let $A'B'$ be the new position of AB. By a continuity argument, $A'n < nB'$ and thus $area(\triangle AnA') < area(\triangle BnB')$, producing a contradiction to the fact that the area of $ABCD$ is minimum. (See Fig. 18.)

(b) According to Lemma 5.3, BC is tangent to Q_i. Let m (respectively, n) be the common point of BC and Q_i closest to B (respectively, C). (Note that n and C may be identical, or that m and n may be identical.) Assume that the midpoint of BC does not lie between m and n. Assume without loss of generality that $Cm < mB$. By rotating BC clockwise around m by an infinitesimal angle θ to a new position $B'C'$ and using the same continuity argument as in (a), we get that the area of quadrilateral $AB'C'D$ is larger than the area of $ABCD$, producing a contradiction. (See Fig. 19.) □

LEMMA 5.5. *A nonconvex polygon P need not have a nondegenerate minimum-area concave quadrilateral.*

Proof. Consider the polygon P formed by taking a large equilateral triangle xyz of side length L and cutting out a small equilateral notch from edge yz of side length ϵ to form a hexagon $uvwxyz$ with reflex angle at v (see Fig. 20). Let $ABCD$ represent a minimum concave quadrilateral containing P with vertex C inside uvw, BC tangent to P at w and CD tangent to P at u. Clearly quadrilateral vertex A should lie above line

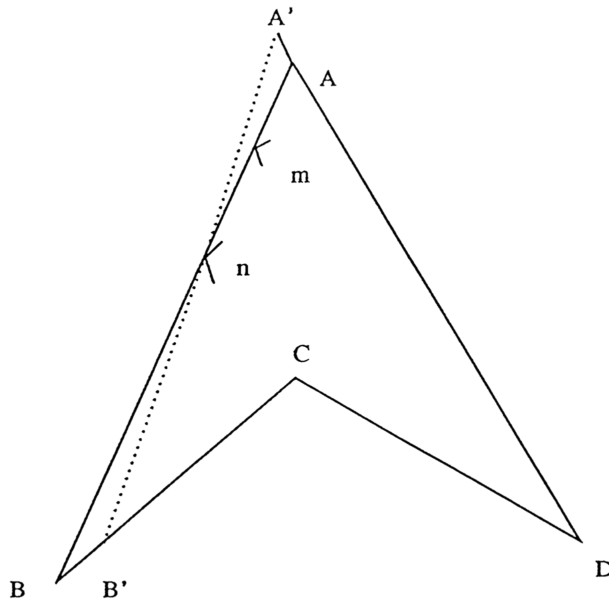


FIG. 18. *The midpoints of AB and AD should lie on $CH(P)$.*

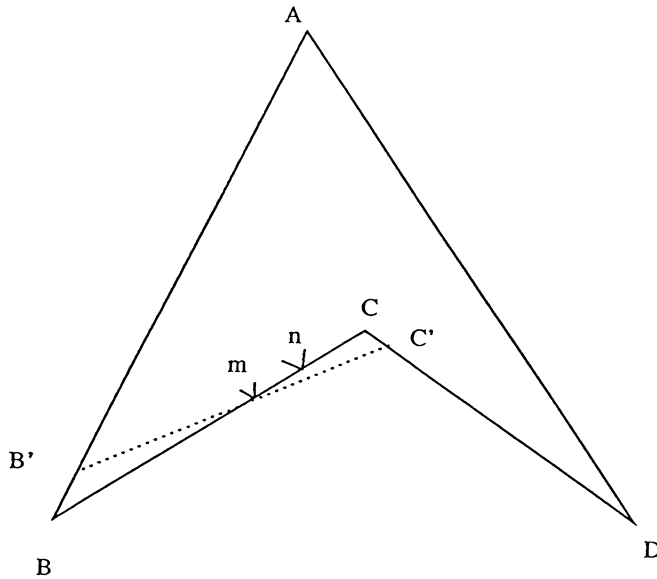


FIG. 19. *The midpoint of BC should lie between m and n.*

yz. According to Lemma 5.4, vertex B (respectively, D) should lie in a half disk with center vertex u (respectively, w) and radius ϵ. By choosing $\epsilon \ll L$, the tangents from B and D to the $CH(P)$ cannot intersect above line yz. Thus a minimum-area concave quadrilateral does not exist. □

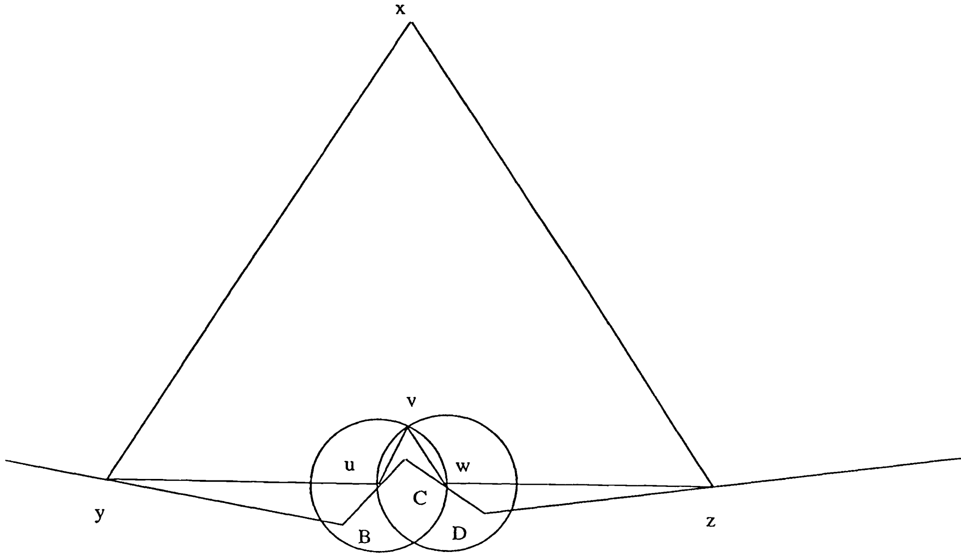


FIG. 20. *A case where an optimum nondegenerate quadrilateral does not exist.*

This characterization of the optimum concave quadrilateral leads to the following algorithm:

Compute $CH(P)$ and let $p =$ the number of pockets.
Triangulate P and all its pockets Q_i, for $i = 1 \cdots p$.
For all pairs k, l of vertices of $CH(P)$ do
  for $i = 1$ to p do begin
    Compute the visibility polygon VQ_i of Q_i from pseudoedge $v_i w_i$ and the shortest path maps inside VQ_i from both v_i and w_i [26].
    Merge those maps [27] and label each region (≤ 6 sides) with its anchors with respect to v_i and w_i.
    For every region R, call $pocket(a, b, k, l, i, R)$.
  end.
Report the optimum.

The procedure $pocket(a, b, k, l, i, R)$ must solve the following optimization problem:

> **Problem.** Construct the minimum area concave quadrilateral $ABCD$ such that (a) AB and AD pass through given points k and l, respectively, (b) the slope of the line through AB (respectively, AD) lies in the interval defined by the slopes of the lines that contain the edges of $CH(P)$ adjacent to k (respectively, l), (c) CB and CD pass through a and b respectively, and (d) $C \in R$.

Solution. Let (a_1, a_2), (b_1, b_2), (k_1, k_2), (l_1, l_2) be the coordinates of points a, b, k, l (as defined previously), respectively. Also let (x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4) be the coordinates of the vertices of the quadrilateral A, B, C, D, respectively. Then the area of the quadrilateral is given by

$$AREA = (x_1 y_2 - x_2 y_1) + (x_2 y_3 - x_3 y_2) + (x_3 y_4 - x_4 y_3) + (x_4 y_1 - x_1 y_4). \tag{11}$$

AB contains k:

$$x_1y_2 - x_2y_1 = k_1y_2 - k_1y_1 + k_2x_1 - k_2x_2. \tag{12}$$

BC contains a:

$$x_3y_2 - x_2y_3 = a_1y_2 - a_1y_3 + a_2x_3 - a_2x_2. \tag{13}$$

CD contains b:

$$x_4y_3 - x_3y_4 = b_1y_3 - b_1y_4 + b_2x_4 - b_2x_3. \tag{14}$$

AD contains l:

$$x_4y_1 - x_1y_4 = l_1y_1 - l_1y_4 + l_2x_4 - l_2x_1. \tag{15}$$

Substituting (12), (13), (14), and (15) into (11), we get

$$\begin{aligned} AREA = {} & (l_1 - k_1)y_1 + (k_1 + a_1)y_2 + (b_1 - a_1)y_3 - (l_1 + b_1)y_4 \\ & +(k_2 - l_2)x_1 - (k_2 + a_2)x_2 + (a_2 - b_2)x_3 + (b_2 + l_2)x_4, \end{aligned} \tag{16}$$

which is linear in terms of the unknown coordinates of A, B, C, D.

Thus we have to solve a linear program in four variables which is subject to a constant number of linear constraints. That problem can be solved in $O(1)$ time. Thus, each pass through the loop above takes $O(n_i + k_i)$ time, where n_i is the size of Q_i and k_i is the size of the subdivision created by merging the shortest path maps from v and w. Thus, if n_c is the number of vertices of $CH(P)$, n_p is the total number of vertices of all pockets of P, and k is the sum of the merged subdivisions over all pockets, then we have the following theorem

THEOREM 5.6. *The minimum concave quadrilateral that contains a simple polygon P can be found either in $O(n_c^2(n_p + k))$ time and $O(n + k)$ space or in $O(n_c^2 n_p^2)$ time and $O(n)$ space.*

Proof. Computing the visibility polygon and the shortest paths within a pocket can be done in $O(n)$ time [26]. Two convex plane subdivisions S_1 and S_2 with m and n vertices, respectively, can be merged in $O(m + n + k)$ time and space, where k is the size of the resultant subdivision. It should be clear that k can range from $O(m + n)$ up to $O(mn)$.

Instead of explicitly merging the two maps, however, we can take each pair (r_1, r_2), where r_1 (respectively, r_2) is a region of the shortest path map from v_i (respectively, w_i), and calculate the intersection region explicitly. For each such intersection region, call $pocket(a, b, k, l, i, R)$. Since every region of the shortest path map is a triangle, the intersection of two such regions has a constant number of sides. The space required is the space to keep the two shortest path maps, i.e., linear. □

The constraints of the linear program do not dictate that vertex C be reflex; the interior angle at C is constrained merely to be greater than or equal to 180 degrees. Thus our algorithm can return degenerate quadrilaterals as solutions. In computing the global minima, we keep the minimum-valued quadrilateral of our local optima, degenerate or otherwise. In cases of a tie, we give precedence to a nondegenerate quadrilateral. If the total minimum is nondegenerate, it solves the global problem. If the total minimum is degenerate, then there is no minimum strictly concave circumscribed quadrilateral for P.

Remark. In case of splinegons a similar technique is applicable. The difference is that the elementary optimization problems are not linear programs anymore, since the sides of the quadrilateral do not pass through constant points and Lemma 5.4 is no longer applicable. Although these elementary continuous optimization problems have a constant number of constraints and a constant number of variables, they are much more complicated.

6. Contained triangles. The maximum area triangle $T = xyz$ contained in a simple polygon P may have $0, 1, 2$, or 3 vertices on the boundary of P. The case of 3 vertices of the triangle on the boundary of P corresponds to the maximum-area inscribed triangle problem solved in a previous section. We focus here on what we call the 0-case, 1-case, and 2-case. To solve these three cases, we use the following lemma.

LEMMA 6.1. *Let A and C (respectively, B and D) be two points on $\overrightarrow{O}x$ (respectively, $\overrightarrow{O}y$) such that segments AB and CD intersect inside the wedge defined by Ox and Oy at point E. Let FG be a line segment through E with F (respectively, G) between A and C (respectively, D and B). The area of the triangle OFG is maximized when one of the following holds: $F = A$ and $G = B$; or $F = C$ and $G = D$ (see Fig. 21(a)).*

Proof. Assume that the maximum occurs when F is between A and C. Assume that $FE < EG$. We can rotate FG by a small angle θ in position $F'G'$ such that $F'E < EG'$. That implies that the area of triangle EGG' is greater than the area of triangle EFF', which in turn implies that the area of OFG is less than the area of $OF'G'$, a contradiction. □

COROLLARY 6.2. *Let $\overrightarrow{O}x$ and $\overrightarrow{O}y$ be two rays with common origin O. Also let A and B be two points on $\overrightarrow{O}x$ and $\overrightarrow{O}y$, respectively, and let C_{AB} be a convex chain as in Fig. 21(b). Let D (respectively, E) lie on OA (respectively, OB) such that O and C_{AB} do not lie on the same side of the line through D and E. Then the area of triangle ODE becomes maximum if DE contains an edge of the convex chain C_{AB}.*

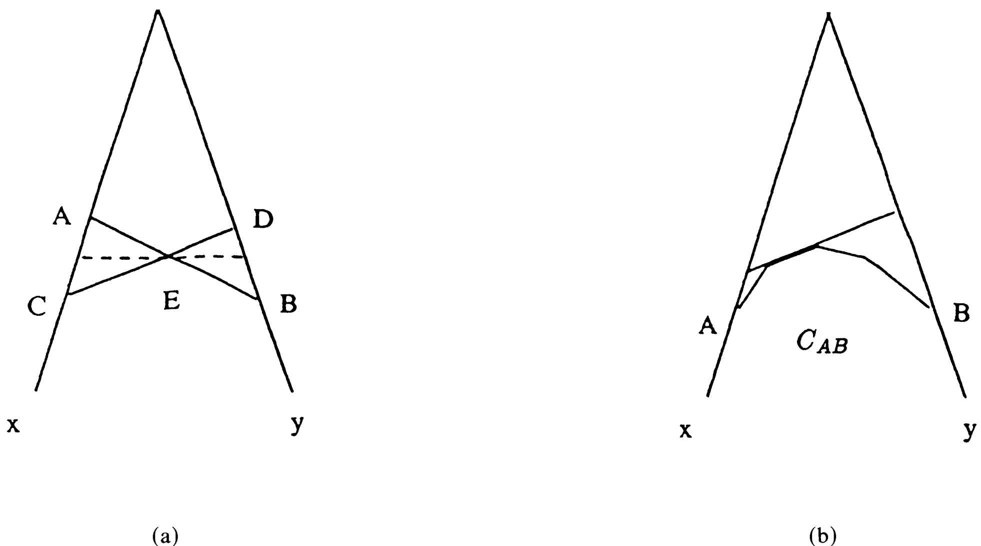


FIG. 21. (a) *Figure for Lemma* 6.1; (b) *Figure for Corollary* 6.2.

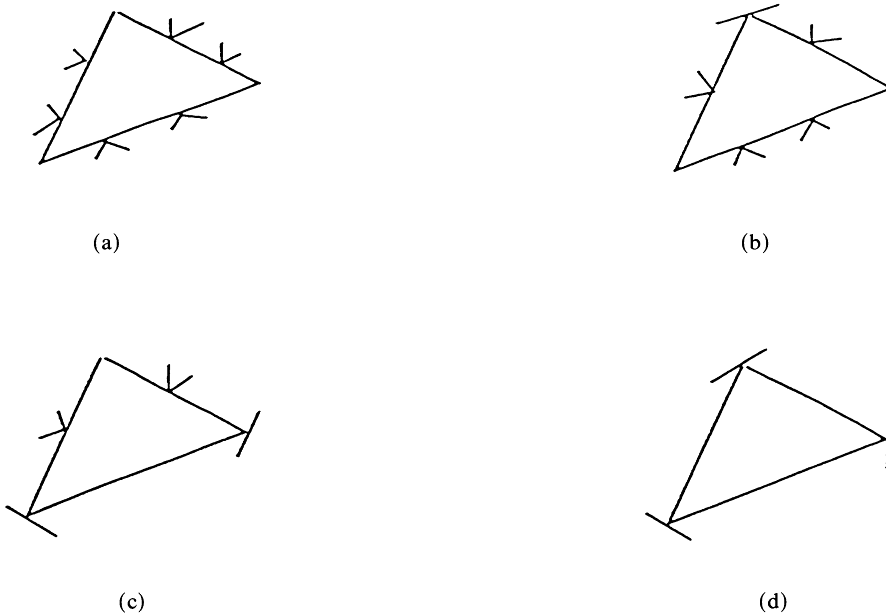


FIG. 22. (a) 0-*case*; (b) 1-*case*; (c) 2-*case*; (d) 3-*case contained triangle.*

Proof. Assume that DE does not contain any edge of C_{AB}. Then either (a) the intersection of DE and C_{AB} is one vertex of C_{AB} or (b) it is the empty set. In the case of (a) we have an instance of Lemma 6.1. In the case of (b) we can translate DE in a direction perpendicular to itself until it intersects C_{AB} and then apply Lemma 6.1. □

LEMMA 6.3. *Let T be a maximum area triangle. Then each edge of T contains at least two points of the boundary of P. Specifically, the following conditions hold (see Fig. 22): If T is of the* 0-*case, then each edge of T contains at least two reflex vertices of polygon P; if T is of the* 1-*case with x on the boundary of P, then yz touches at least two reflex vertices of P and xy and xz at least one; if T is of the* 2-*case with y and z on edges i and j, there exists at least one edge k of P such that i, j, k define a fan-shaped polygon $F_{i,j,k} \supseteq T$.*

Proof. Assume that there exists at least one side of the triangle xyz for which the above argument is not true. Without loss of generality, assume yz is that edge. Extend both xy and xz to the side of y and z, respectively, until they intersect the boundary of P. Let v and w be the two intersection points. Consider the convex hull of the part of P that is between v and w and inside the triangle xvw. Then applying the previous corollary, we can move yz so that the area of xyz increases. □

This characterization of the 0-, 1-, 2-case maximum area triangle contained in a simple polygon suggests algorithms for finding these triangles.

0-case triangle algorithm. One algorithm would consider all triples of pairs of reflex vertices i.e., $O(n^6)$ objects; check whether the corresponding triangle is contained in P in $O(\log n)$ time using ray shooting [13] or [26]; and choose the largest one. This brute force approach requires $O(n^6 \log n)$ arithmetic operations. Another less naive algorithm would fix two sides of the candidate triangle by choosing a couple of pairs of reflex vertices (A, B) and (E, F), assuming that xy contains AB and xz contains EF, and spend linear time to find the optimum position of yz, for a total of $O(n^5)$ operations.

Using the linearity of shortest path trees inside simple polygons, we can reduce the complexity by an order of magnitude (see Fig. 23). Fix a pair of reflex vertices C and D with the characteristic that $\overline{CD} \subset P$ and all edges incident to C and D lie on the same side of the line containing $\overline{CD}$. Assume that side yz contains these vertices. Determine in $O(\log n)$ time ([13] or [26]) the points G and H closest to C and D, respectively, where the line through segment CD intersects the boundary of P. Let P' represent the subpolygon of P that lies at the opposite side of GH from the edges incident to C and D. Since x must be visible from GH, the shortest paths from G to x and from H to x inside P must be inward convex chains containing segments AB and EF, respectively. That implies AB (respectively, EF) are edges of the shortest path tree from G (respectively, H) inside P'. Since the sizes of the shortest path trees are linear in the size of P' and therefore in the size of P, we need consider only pairs of the $O(n)$ edges of the shortest path tree from G and the $O(n)$ edges of the shortest path tree from H, a total of $O(n^2)$ objects.

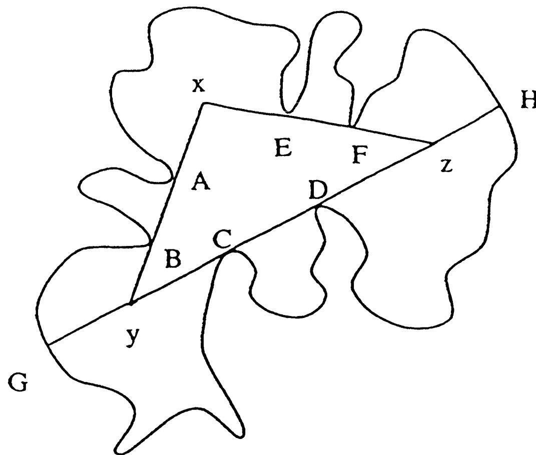


FIG. 23. 0-*case contained triangle.*

How can we test efficiently whether the chosen pair AB and EF of shortest path tree edges forms a legal triangle with GH? Choose only those pairs AB such that (a) the shortest paths from G to A and G to B are inward convex chains, (b) points y and z do not lie in segment CD, and (c) segments Ax and Bx lie inside P. We need the comparable conditions for EF. Conditions (a) and (b) clearly can be checked in $O(1)$ time. One way to test condition (c) is to apply ray shooting inside P in $O(\log n)$ time. Since AB and EF are shortest-path tree edges, however, constant time suffices. Define e_1 (respectively, e_2) as the edge of the shortest path map from G which is adjacent to vertex A (respectively, E) and collinear with AB (respectively, EF). If e_1 and e_2 both exist and intersect, then the intersection point is a valid vertex x. Repeating the above procedure for every one of the $O(n^2)$ pairs of reflex vertices C and D yields the following lemma.

LEMMA 6.4. *The* 0-*case maximum triangle can be found in* $O(n^4)$ *time and* $O(n)$ *space.*

1-case triangle algorithm. As in the 0-case algorithm, we fix a pair of reflex vertices C and D (see Fig. 24). We again find points G and H as defined previously. We then have to walk on the shortest path maps of G and H along the boundary of P, as we did in the inscribed triangle case of §4. Thus for a fixed pair of reflex vertices we spend, using similar arguments, $O(n)$ time and therefore a total $O(n^3)$ for the whole problem.

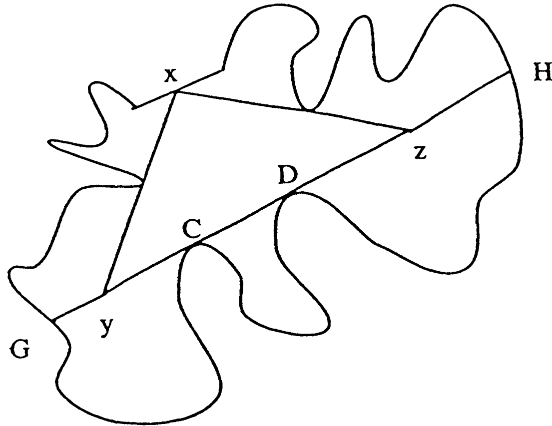


FIG. 24. *1-case contained triangle.*

LEMMA 6.5. *The* 1-*case maximum triangle can be found in* $O(n^3)$ *time and* $O(n)$ *space.*

2-case triangle algorithm. According to Lemma 6.3, triangle xyz lies in a fan-shaped polygon where y (respectively, z) lies on edge i (respectively, j) (see Fig. 25). Subdivide j (respectively, i) according to the shortest path maps from both p_{i+1}, p_i and p_{k+1} (p_j, p_{j+1}, and p_k, respectively). For each interval on the subdivision of edge i and each interval of the subdivision of edge j, (a) check whether points y and z are visible, using techniques developed in §4, (b) let $a = anchor^{p_{k+1}}(z)$ and $b = anchor^{p_k}(y)$. Then, check whether $SP_{p_{k+1}}(a)$ and $SP_{p_k}(b)$ are inward convex. (c) Check whether the intersection x of the lines through segments yb and za lies "below" the line through edge k. That guarantees that triangle xyz lies inside $F_{i,j,k}$. (d) Solve the appropriate continuous optimization problem. It should be clear that steps (a) through (d) take $O(1)$ time.

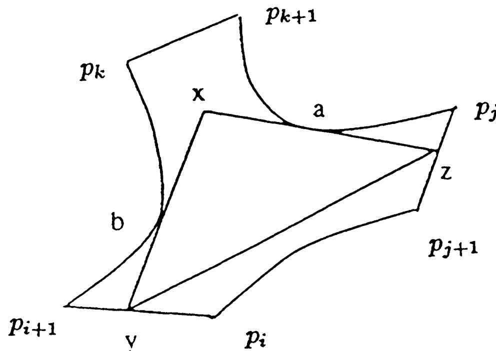


FIG. 25. *2-case contained triangle.*

According to the above discussion, the time complexity per fan-shaped polygon is $O((s_j^{p_{i+1}} + s_j^{p_k})(s_i^{p_j} + s_i^{p_{k+1}}))$. Then summing over all fan-shaped polygons we get

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} O((s_j^{p_{i+1}} + s_j^{p_k})(s_i^{p_j} + s_i^{p_{k+1}})) = O(n^3).$$

Given that the total shortest path computation takes $O(n^4)$ time and $O(n)$ space, we have the following lemma.

LEMMA 6.6. *The 2-case maximum triangle can be found in* $O(n^4)$ *time and* $O(n)$ *space.*

THEOREM 6.7. *The maximum area triangle contained in a simple polygon can be found in* $O(n^4)$ *time and* $O(n)$ *space.*

Unfortunately, solving this problem for splinegons seems complicated. All that we know is that each side of the maximum area triangle contained in a simple splinegon touches the splinegon boundary on at least one point. This constraint is not enough to produce anything other than a "brute force" algorithm.

7. Conclusions. We solved various geometric optimization problems using shortest paths within simple closed regions. There are several directions for further research. One is the obvious task of improving the current time bounds and extending our results in higher dimensions. Another question is whether shortest paths can be used in other inclusion, enclosure, or separator problems, or, more interestingly, whether they can be used in other classes of geometric optimization problems. A more exhaustive study of optimization over curved objects would also be appropriate.

Acknowledgments. The authors gratefully acknowledge the careful reading of the two anonymous referees and their many suggestions for improving the readability of the paper.

REFERENCES

[1] A. AGGARWAL, *Lecture notes in computational geometry*, MIT Research Seminar Series MIT/LCS/RSS 3, August, 1988.

[2] A. AGGARWAL, J. S. CHANG, AND C. K. YAP, *Minimum area circumscribing polygons*, Visual Comput., 1 (1985), pp. 112–117.

[3] A. AGGARWAL, M. KLAWE, S. MORAN, P. SHOR, AND R. WILBER, *Geometric applications to a matrix searching algorithm*, Algorithmica, 2 (1987), pp. 209–233.

[4] A. AGGARWAL AND J. PARK, *Notes on searching in multidimensional monotone arrays*, in Proceedings of the 29th IEEE Symposium on Foundations of Computer Science, 1988, pp. 497–512.

[5] J. E. BOYCE, D. P. DOBKIN, R. L. DRYSDALE, AND L. J. GUIBAS, *Finding Extremal Polygons*, SIAM J. Comput., 14 (1985), pp. 134–147.

[6] R. D. BOURGIN AND S. E. HOWE, *Algorithms for shortest curves in planar regions with curved boundary*, manuscript, 1989.

[7] R. D. BOURGIN, M. S. MARTIN, AND P. L. RENZ, *Shortest curves in Jordan regions vary continuously with the boundary*, Adv. Math., to appear.

[8] R. D. BOURGIN AND P. L. RENZ, *Shortest paths in simply connected regions in* R^2, Adv. Math., 76 (1989), pp. 260–295.

[9] J. S. CHANG AND C. K. YAP, *A polynomial solution for potato-peeling and other polygon inclusion and enclosure problems*, Discrete Comput. Geom., 1 (1986), pp. 155–182.

[10] J. S. CHANG, *Polygon optimization problems*, Ph.D. thesis, Department of Computer Science, New York University, New York, NY, 1986.

[11] B. CHAZELLE, *Triangulating a simple polygon in linear time*, Proceedings of the 31st IEEE Symposium on Foundations of Computer Science, 1990, pp. 220–230.

[12] B. CHAZELLE AND D. DOBKIN, *Intersection of convex objects in two and three dimensions*, J. Assoc. Comput. Mach., 34 (1987), pp. 1–27.

[13] B. CHAZELLE AND L. GUIBAS, *Visibility and intersection problems in plane geometry*, Discrete Comput. Geom., 4 (1989), pp. 551–581.

[14] P. CHEW AND K. KEDEM, *Placing the largest similar copy of a convex polygon among polygonal obstacles*, Proceedings of the Association of Computing Machinery, Symposium on Computational Geometry, 1989, pp. 167–174.

[15] N. A. DEPANO, *Polygon approximation with optimized polygonal enclosures: applications and algorithms*, Ph.D. thesis, Dept. of Computer Science, Johns Hopkins University, Baltimore, MD, April 1988.

[16] N. A. DEPANO, Y. KE, AND J. O'ROURKE, *Finding largest inscribed equilateral triangles and squares*, in Proceedings of the Allerton Conference, 1987.

[17] D. DOBKIN AND L. SNYDER, *On a general method for maximizing and minimizing among certain geometric problems*, in Proceedings of the 20th IEEE Symposium on Foundations of Computer Science, 1979 , pp. 9–17.

[18] D. DOBKIN AND D. SOUVAINE, *Computational geometry in a curved world*, Algorithmica, 5 (1990), pp. 421–457.

[19] D. DOBKIN, D. SOUVAINE, AND C. VAN WYK, *Decomposition and intersection of splinegons*, Algorithmica, 3 (1988), pp. 473–485.

[20] H. EDELSBRUNNER, L. GUIBAS, AND G. STOLFI, *Optimal point location in monotone subdivisions*, SIAM J. Comput., 15 (1986), pp. 317–340.

[21] R. FLEISCHER, K. MEHLHORN, G. ROTE, E. WELZL, AND C. YAP, *On simultaneous inner and outer approximation of shapes*, in Proceedings of the 6th ACM Symposium on Computational Geometry, June 1990, pp. 216–224.

[22] S. FORTUNE, *A fast algorithm for polygon containment by translation*, Proceedings of the 13th International Colloquium on Automata, Languages and Programming 1985 , pp. 189–198.

[23] A. FOURNIER AND D. Y. MONTUNO, *Triangulating simple polygons and equivalent problems*, ACM Trans. Graphics, 3 (1984), pp. 153–174.

[24] M. GAREY, D. JOHNSON, F. PREPARATA, AND R. TARJAN, *Triangulation of a simple polygon*, Inform. Process. Lett., 7 (1978), pp. 175–179.

[25] L. GUIBAS AND J. HERSHBERGER, *Optimal shortest path queries in a simple polygon*, in Proceedings of the 3rd ACM Symposium on Computational Geometry, 1987, pp. 50–63.

[26] L. GUIBAS, J. HERSHBERGER, D. LEVEN, M. SHARIR, AND R. TARJAN, *Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons*, Algorithmica, 2 (1987), pp. 209–233.

[27] L. GUIBAS AND R. SEIDEL, *Computing convolutions via reciprocal search*, Discrete Comput. Geom., 2 (1988), pp. 175–193.

[28] L. GUIBAS AND J. STOLFI, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams*, ACM Trans. Graphics, 4 (1985), pp. 74–123.

[29] J. HERSHBERGER, *An optimal visibility graph algorithm for triangulated simple polygons*, Algorithmica, 4 (1989), pp. 141–155.

[30] V. KLEE AND M. LASKOWSKI, *Finding the smallest triangles containing a given convex polygon*, J. Algorithms, 6 (1985), pp. 359–375.

[31] D. T. LEE AND F. P. PREPARATA, *Euclidean shortest paths in the presence of rectilinear barriers*, Networks, 14 (1984), pp. 393–410.

[32] B. LISPER, *Theory Net posting and followup communication*, July, 1988.

[33] E. A. MELISSARATOS, *Mesh generation and geometric optimization*, Ph.D. thesis, in preparation, Rutgers University, New Brunswick, NJ, 1991.

[34] E. A. MELISSARATOS AND D. L. SOUVAINE, *On solving geometric optimization problems using shortest paths*, in Proceedings of the 6th ACM Symposium on Computational Geometry, June 1990, pp. 350–359.

[35] ———, *Shortest paths, visibility, and optimization problems in planar curvilinear objects*, in Proceedings of the 2nd Canadian Conference on Computational Geometry, August 1990, pp. 337–342.

[36] J. D. MITTLEMAN AND D. L. SOUVAINE, *Shortest area-bisector of a convex polygon*, Technical Report LCSR-TR-139, Laboratory for Computer Science Research, Rutgers University, New Brunswick, NJ, November 1989.

[37] J. O' ROURKE, A. AGGARWAL, S. MADDILA, AND M. BALDWIN, *An optimal algorithm for finding minimal enclosing triangles*, J. Algorithms, 7 (1986), pp. 258–269.

[38] M. H. OVERMARS AND J. VAN LEEUWEN, *Maintenance of configurations in the plane*, J. Comput. System Sci., 23 (1981), pp. 166–204.

[39] O. SCHWARZKOPF, U. FUCHS, G. ROTE, AND E. WELZL, *Approximation of convex figures by pairs of rectangles*, in Proceedings of the Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 415, 1990, pp. 240–249.

[40] A. SHAFFER AND C. J. VAN WYK, *Convex hulls of piecewise-smooth Jordan curves*, J. Algorithms, 8 (1987), pp. 66–94.

[41] D. L. Souvaine, *Computational geometry in a curved world*, Ph.D. thesis, Department of Computer Science, Princeton University, Princeton, NJ, October, 1986.

[42] R. E. Tarjan and C. Van Wyk, *Triangulation of a simple polygon*, SIAM J. Comput., 17 (1988), pp. 143–178.

SIAM J. COMPUT.
Vol. 21, No. 4, pp. 639-654, August 1992
© 1992 Society for Industrial and Applied Mathematics
002

DYNAMIC TREE EMBEDDINGS IN BUTTERFLIES AND HYPERCUBES*

F. T. LEIGHTON†, MARK J. NEWMAN‡, ABHIRAM G. RANADE§, AND ERIC J. SCHWABE¶

Abstract. This paper presents simple randomized algorithms for dynamically embedding M-node binary trees in either a butterfly or a hypercube network of N processors. These algorithms are *dynamic* in the sense that the tree to be embedded may start as one node and grow by dynamically spawning children. The nodes are incrementally embedded as they are spawned. Thus, the algorithm is especially suited for maintaining dynamic tree structures like those in divide-and-conquer and branch-and-bound algorithms.

In the embeddings, the paper seeks to optimize the *load* on the processors of the network, the *dilation* of the tree edges, and the *congestion* on the network edges, in order to satisfy the demands of load balancing, process locality, and communication efficiency. The paper begins by presenting a simple level-by-level scheme for dynamically embedding trees in a butterfly network, and by successive modifications. The following results are obtained:

1. An embedding algorithm for the hypercube that achieves dilation 1 and, with high probability, load $O((M/N)+\log N)$.
2. An embedding algorithm for the butterfly that achieves dilation 2 and, with high probability, load $O((M/N)+\log N)$.
3. An embedding algorithm for the hypercube that achieves dilation $O(1)$ and, with high probability, load $O((M/N)+1)$ and congestion $O((M/N)+1)$.

The third embedding simultaneously optimizes load and dilation to within constant factors, optimizing congestion as well when $M=O(N)$. The first two embeddings are also optimal to within constant factors when the tree to be embedded is large (i.e., $M=\Omega(N\log N)$).

In addition, this paper proves a lower bound of $\Omega(\sqrt{\log N})$ dilation for deterministic embedding algorithms that achieve load $O((M/N)+1)$, which implies that any embedding algorithm that simultaneously optimizes load and dilation must be randomized.

Key words. dynamic embeddings, binary trees, hypercube, butterfly network

AMS(MOS) subject classifications. 05C10, 68Q99, 69R05, 69R10, 94C15

1. Introduction. Achieving high performance on a parallel computer requires the satisfaction of two potentially conflicting requirements. First, the computational load posed by the program should be evenly shared among all processors (load balancing). Second, processes communicating frequently should be placed on processors that are close (communication locality).

This problem has been studied abstractly as the problem of embedding a process graph G in a processor graph H [2]-[7], [10]. The vertices of G are processes comprising

* Received by the editors November 14, 1990; accepted for publication (in revised form) August 5, 1991. This research was supported by the U.S. Air Force under Air Force Office of Scientific Research contracts 86-0078 and 89-0271, the Defense Advanced Research Projects Agency under contracts N00014-80-C-0622, N00014-87-K-825, and N00014-89-J-1988, the U.S. Army under contract DAAL03-86-K-0171, a National Science Foundation Presidential Young Investigator Award with matching funds from International Business Machines, Inc., a National Science Foundation Graduate Fellowship and a Defense Advanced Research Projects Agency/National Aeronautics and Space Administration Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland. It was performed while the second and fourth authors were members of the Mathematics Department and Laboratory for Computer Science, Massachusetts Institute of Technology. A preliminary version of this paper appeared in the Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures, pp. 224-234.

† Mathematics Department and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.

‡ Temple, Barker & Sloane, Inc., 33 Hayden Avenue, Lexington, Massachusetts 02173.

§ Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720.

¶ School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

the parallel program, with edges representing communication between processes. The vertices of H are processors, and the edges represent communication channels. For many computations, it is possible to predict G before execution. In such cases it is useful to map the vertices of G into those of H so as to minimize certain parameters. The first is the maximum *load* per processor, i.e., the maximum number of processes placed on any processor. The second is the *dilation*, i.e., the maximum distance between the images in H of pairs of processes that are neighbors in G. Another parameter sometimes considered is the *congestion* of the embedding, i.e., the maximum number of times an edge of H is "traced over" by paths corresponding to the edges of G.

This paper focuses on embedding arbitrary binary trees into butterfly and hypercube networks. Trees arise naturally in many computations: divide-and-conquer algorithms, branch-and-bound search [9], functional expression evaluation, and image understanding (quad/oct trees). Bhatt, Chung, Leighton, and Rosenberg [3] showed that every N-node binary tree could be embedded in an N-processor hypercube such that each processor received a single tree node, and the maximum dilation was $O(1)$. Embedding trees into butterfly networks is harder, because the butterfly is much sparser than the hypercube. Bhatt, Chung, Hong, Leighton, and Rosenberg [2] later showed how to embed the complete binary tree with N nodes in a butterfly network with N processors with constant dilation and load. The problem of embedding arbitrary trees into butterfly networks was left open.

Tree-structured computations are often dynamic. As the computation progresses, the tree may grow or shrink in a manner that may be impossible to predict beforehand. Bhatt and Cai [1] proposed a dynamic version of the embedding problem. They considered a process graph that is a binary tree that can grow during execution. At each step any node of the tree that does not have two children can request to spawn a child. The dynamic embedding problem is harder than the static one because newly spawned children must be allocated to processors incrementally, without making assumptions about how the tree will grow in the future. Further, the placement decision must itself be implemented within the network in a distributed manner without accessing global information. The paradigm proposed by Bhatt and Cai disallows process migration; i.e., once a process is placed on a particular processor, it cannot be moved subsequently. Obviously, allowing migration can potentially give better load balancing/dilation but can also be extremely expensive in practice.

Bhatt and Cai [1] presented a randomized algorithm for dynamically growing trees with M vertices on an N processor binary hypercube. Each child process is placed no farther than distance $O(\log \log N)$ from its parent. Furthermore, with high probability[1] (independent of the shape of the tree) the algorithm assigns only $O(M/N + 1)$ vertices to each processor. The congestion of the embedding was not determined but is probably on the order of $\log^{\alpha} N$ for some constant α.

1.1. SUMMARY OF RESULTS. We consider the problem of growing trees on butterfly and hypercube networks. Our framework is identical to that of Bhatt and Cai [1], although our growth algorithms are substantially simpler and have provably better performance. We begin by describing a level-by-level strategy for embedding a binary tree in a butterfly. Modifications to this scheme form the basis of all our embedding algorithms. The first modification we introduce is the use of random *flip bits*, which

[1] We use the phrase "Q is less than $O(f)$ with high probability" to mean, "For every c there exists a constant k independent of N such that the probability that Q exceeds kf is less than N^{-c}."

randomize the locations of tree nodes within a level of the butterfly. Analysis of the behavior of these flip bits is sufficient to prove our first result.

THEOREM 1. *An arbitrary binary tree T with M vertices can be dynamically grown on an N processor hypercube with dilation* 1 *such that with high probability the maximum load per processor is* $O(M/N+\log N)$.

Note that this is optimal to within a constant factor whenever the tree T is large (i.e., $M=\Omega(N\log N)$). For these large trees, it gives an optimal $O(M/N)$ load, as did Bhatt and Cai [1], while improving the dilation from $O(\log\log N)$ to 1. Next we present another modification of the scheme involving *level balancing*—in effect, we stretch certain paths within the tree so that the number of tree nodes assigned to any level of the butterfly is balanced. This modification leads to our next result, this time for a butterfly.

THEOREM 2. *An arbitrary binary tree T with M vertices can be dynamically grown on an N processor butterfly network with dilation* 2 *such that with high probability the maximum load per processor is at most* $O(M/N+\log N)$.

Again, this is optimal to within a constant factor when $M=\Omega(N\log N)$. This result is a substantial improvement over previous work since not even good static embeddings of arbitrary binary trees were previously known. Finally, we take advantage of an embedding of the butterfly into the hypercube that embeds entire levels of the butterfly to subcubes of the hypercube in order to develop a scheme for local redistribution of load within levels. This leads to an embedding algorithm for the hypercube that simultaneously optimizes maximum load and dilation. In addition, the congestion of the embedding is optimal if $M=O(N)$.

THEOREM 3. *An arbitrary binary tree T with M vertices can be dynamically grown on an N processor hypercube with constant dilation such that with high probability the maximum load per processor is* $O(M/N+1)$ *and the maximum congestion on any edge is* $O(M/N+1)$.

It should be noted that although our theorems are phrased in terms of trees that only grow, these embedding algorithms are also effective for dynamic trees that can both grow and shrink at their leaves. Consider a binary tree T that grows and shrinks. At each stage in the tree's evolution, the probability space of possible embeddings of the current form of the tree T' is equivalent to the space of embeddings that would have occurred had we simply grown the tree T' by using the same algorithm. Therefore, the same results hold for each step in the tree's evolution (assuming, of course, that the total number of steps in the tree's evolution is bounded by a polynomial in N).

We also prove a lower bound for deterministic embedding algorithms for hypercubes that shows that any deterministic algorithm that balances load must necessarily have dilation $\Omega(\sqrt{\log N})$. It follows that any embedding algorithm that simultaneously optimizes load and dilation (to within constant factors) must be randomized. This consequence also holds for the butterfly, since it is a subgraph of the hypercube [6].

1.2. OVERVIEW. The basic embedding algorithm is presented in § 2, along with the introduction of flip bits and the proof of Theorem 1. The level-balancing scheme is introduced and analyzed in § 3, along with the proof of Theorem 2. Improvements to the hypercube embedding algorithm and the proof of Theorem 3 are given in § 4. Section 5 states and proves the lower bound for deterministic algorithms.

2. The basic growth algorithm. We use a butterfly network with $n2^n$ processors organized as n levels, each consisting of 2^n processors. Each processor in the butterfly is assigned a unique label $\langle l, c\rangle$, where $0\leqq l<n$, $0\leqq c<2^n$. Processors $\langle l, *\rangle$ constitute

the lth level. Processors $\langle *, c\rangle$ constitute the cth column. Processor $\langle l, c\rangle$ is connected to processors $\langle l+1 \bmod n, c\rangle$ and to processor $\langle l+1 \bmod n, c\oplus 2^l\rangle$, where $\oplus$ denotes bitwise exclusive OR. Notice that level $n-1$ is connected to level 0, so that each column of the butterfly wraps around to become a cycle.

2.1. PRELIMINARY SCHEME. We begin with a *level-by-level* strategy for growing a tree on an N-node butterfly network.

In the cases in which we are ultimately interested in an embedding in a hypercube, we will first embed the tree in a butterfly and then consider some embedding of the butterfly in the hypercube. We place the root of the tree on processor $\langle 0, 0\rangle$ in the butterfly. This processor is connected to two processors in level 1, on which we place the children of the root. These processors are in turn connected to four level-2 processors, which will in turn receive the children of the root's children, and so on. This strategy enables us to grow any n-level binary tree with dilation 1 and with at most one tree vertex per butterfly processor. Trees with greater height are wrapped around; i.e., level-n vertices are placed in butterfly level 0, level-$n+1$ vertices in butterfly level 1, and so on. The set of tree vertices that are mapped to level i of the n-level butterfly consists of those vertices in levels i, $i+n$, $i+2n\ldots$ of the tree; we refer to this as the ith *level set* of the tree. There are two issues we need to consider:

1. *Evenly distributing tree vertices within each level.* We would like the vertices belonging to level set i to be evenly distributed among the processors in the ith level of the butterfly, i.e., to guarantee that no single processor in level i receives too many vertices.

2. *Evenly distributing tree vertices among different butterfly levels.* For example, when embedding a complete binary tree of height h, level $h-1 \bmod n$ of the butterfly would receive all the leaves of the tree, or nearly half the total number of vertices. Ideally, we would like the vertices to be divided evenly among all the levels of the butterfly.

We will defer our consideration of the second issue until § 3. First, a modification of the basic scheme will help us achieve balance within a level.

2.2. FLIP BITS. A random *flip bit* is generated at each vertex of the tree to decide where its children will be spawned. Consider a vertex v of the tree that has been placed on some processor p in level i of the butterfly. This node is connected to processors q and r in level $i+1 \bmod n$, which will receive the children of v. The flip bit chosen for vertex v decides whether the left child of v will be placed on q or on r. The right child is then placed on the other processor. Note, of course, that it is not necessary that v have two children—the bit only determines where the children will be placed if they are every spawned.

In § 3 we will show that this ensures even distribution with each level. Intuitively, each vertex is effectively placed by using a random path determined by the flip bits chosen along its ancestors. For now, this modified scheme is sufficient to prove Theorem 1.

THEOREM 1. *An arbitrary binary tree T with M vertices can be grown dynamically on an N processor hypercube with dilation 1 such that with high probability the maximum load per processor is $O(M/N+\log N)$.*

The theorem is immediate from the following lemma.

LEMMA 1. *An arbitrary tree T with M vertices can be grown in a butterfly network of N processors such that each column in the butterfly receives no more than $O(M/2^n+n)$ vertices with high probability.*

Suppose this lemma were true. Then, by simulating the $N=n2^n$-node butterfly by a 2^n-node hypercube, where each node of the hypercube simulates an entire column of the butterfly, we have an embedding algorithm for the hypercube that achieves

dilation 1 and load $O(M/N+\log N)$ with high probability. Thus, Lemma 1 is sufficient to prove Theorem 1.

The general idea behind the proof of Lemma 1 is that a large number of vertices will be placed in the same column in the butterfly only if the flip bits on the paths leading to these vertices are chosen in a very specific (and therefore unlikely) manner.

A *stagnant path* p is a maximal path $v(1), v(2), \cdots, v(l)$ in T with $v(1)$ toward the root such that all $v(i)$ are placed in the same column v of the butterfly. Let the *leader* of p be the nth ancestor of $v(1)$, and let the *trace* of p be the set of $n+l-1$ vertices between the leader (inclusive) and $v(l)$ (exclusive). If $v(1)$ is in the first n levels of the tree, then the leader of the path is defined to be the root of the tree.

Notice that there is a unique path in the butterfly from the leader of a stagnant path p to vertex $v(1)$. Thus, given the column in which the leader lies and the column in which the path p lies, we can completely determine the flip bits chosen along the trace of the path. The next observation is that the traces of distinct stagnant paths mapped to the same column are distinct; i.e., the information gained from one trace is different from that obtained in the other.

LEMMA 2. *Let p and p' be two distinct stagnant paths mapped to the same column of the butterfly. Then their traces are vertex disjoint in the tree.*

Proof. Contrary to the lemma, suppose that the lowest point in the tree at which the traces intersect is vertex u. At vertex u, the two traces are mapped to the same column of the butterfly. Likewise, the two stagnant paths are mapped to the same column. The two children of u are mapped to different columns of the butterfly, however, so that the traces must reconverge in some butterfly column between the children of u and the beginnings of the two stagnant paths. However, the two paths cannot meet again in any column until they have traversed all n levels of the butterfly. Since the two stagnant paths are at a distance less than n from u, the traces cannot reconverge in the butterfly before reaching them, and we have a contradiction. □

LEMMA 3. *For any column v of the butterfly, there is at most one stagnant path mapped to v such that $v(1)$ is in the first n levels of the tree.*

Proof. This lemma follows immediately from Lemma 2 because any two such paths will have the same leader—the root of the tree. □

Proof of Lemma 1. We shall count the number of different settings of the flip bits that give rise to some column having at least $C=k(M/2^n+n)$ tree vertices. This can be done as follows:

1. Let C_0 be the number of stagnant paths $(1 \leqq C_0 \leqq C)$, and define $\beta = C/C_0$.
2. Choose the column: 2^n choices.
3. Choose the endpoint of each path: $\binom{M}{C_0}$ choices.
4. Choose the lengths of the paths: $\binom{C+C_0}{C_0}$ choices.
5. Choose the flip bits at all vertices in T except those in the C_0 traces. The total number of flip bits is M, and the length of the jth trace is $n+l_j-1$, except for the possible case when one stagnant path has v_1 in the first n levels of the tree, in which case the length of its trace is l_j-1. Thus, the total number of bits this step fixes is $M-\sum (n+l_j-1)+n = M-(C_0(n-1)+C)+n$. Thus, the total number of choices is $2^{M-(C_0(n-1)+C)+n}$.

First, we claim that the above choices completely determine all the flip bits. To see this, consider the trace with its leader belonging to the smallest level in T, of all traces. Clearly, the last step of the above procedure fixes the position of the leader. This fixes all the bits in the trace, since the endpoint and the length of the trace are known. The bits for the other traces are similarly determined.

The total number of ways of choosing all the bits is 2^M. Summing over the C possible values of C_0, the probability that some column gets more than C vertices is

at most

$$\left(\sum_{C_0=1}^{C} 2^n \binom{M}{C_0}\binom{C+C_0}{C_0} 2^{M-(C_0(n-1)+C)+n}\right)/2^M$$

$$\leqq 2^{2n} \sum_{C_0=1}^{C} \binom{M}{C_0}\binom{C+C_0}{C_0} 2^{-(C_0(n-1)+C)}$$

$$\leqq 2^{2n} \sum_{C_0=1}^{C} \left(\frac{M(C+C_0)\,e^2}{C_0^2}\right)^{C_0} 2^{-(C_0(n-1)+C)}$$

$$\leqq 2^{2n} \sum_{C_0=1}^{C} \left(\frac{2e^2 M(C+C_0)}{C_0^2 2^n 2\beta}\right)^{C_0}$$

$$\leqq 2^{2n} \sum_{C_0=1}^{C} \left(\frac{2e^2}{k}\,\frac{\beta(\beta+1)}{2^\beta}\right)^{C_0}.$$

To go from the second line to the third we have used the inequality $\binom{n}{r} \leqq (ne/r)^r$. Choosing $k > 10e^2$ and noting that for any β, $\beta(\beta+1) \leqq 5(2^{\beta/2})$, we can simplify the above expression to

$$2^{2n} \sum_{C_0=1}^{C} \left(\frac{1}{2^{\beta/2}}\right)^{C_0} \leqq 2^{2n} \sum_{C_0=1}^{C} 2^{-C/2} \leqq 2^{2n} C 2^{-C/2} \leqq 2^{-C/4} \leqq 2^{-kn/4} \leqq N^{-k/8}. \qquad \square$$

3. Embedding in the butterfly. In this section we introduce a modification to the embedding algorithm that ensures that with high probability the nodes of the binary tree are distributed evenly among the levels of the butterfly. We then prove that the flip bits described in the previous section are sufficient to distribute the tree nodes evenly within each level.

3.1. A LEVEL-BALANCING TRANSFORMATION. We transform the tree T being grown by selectively inserting dummy vertices into some of its edges during the growth. Even if some level originally has a disproportionately large number of vertices, the newly introduced vertices will help to even the distribution of the tree vertices among the levels.

The n-way level-balancing transformation is as follows. Define a vertex of T to be *distinguished* if it lies in level $i \equiv 0 \bmod n/3$ of the tree.[2] For each distinguished vertex v in T we pick a random number $S(v)$ between 0 and $n/3$ called the *stretch count.* We insert a single dummy vertex in each of the edges that connect v to its descendants in levels $i+1$ through $i+S(v)$. Figure 1 illustrates the transformation. Note that this transformation can be applied as the tree grows. Each node needs to know only what level of the tree T it belongs to and the stretch count generated at its nearest distinguished ancestor. This information is sufficient to decide whether or not a dummy vertex is inserted when a child is spawned.

The new tree $B(T)$ that results is grown on the butterfly by using the procedure described in § 2.2, which gives a dilation-1 embedding for $B(T)$. This corresponds to a dilation-2 embedding of T, since some of the edges in T were replaced by two edges in $B(T)$.

[2] In what follows we may make references like "(mod x)" or "contribution of x messages" when x may not be integral. Rounding these quantities to integers does not affect the correctness of the proof. For ease of exposition, we shall not consider the issue.

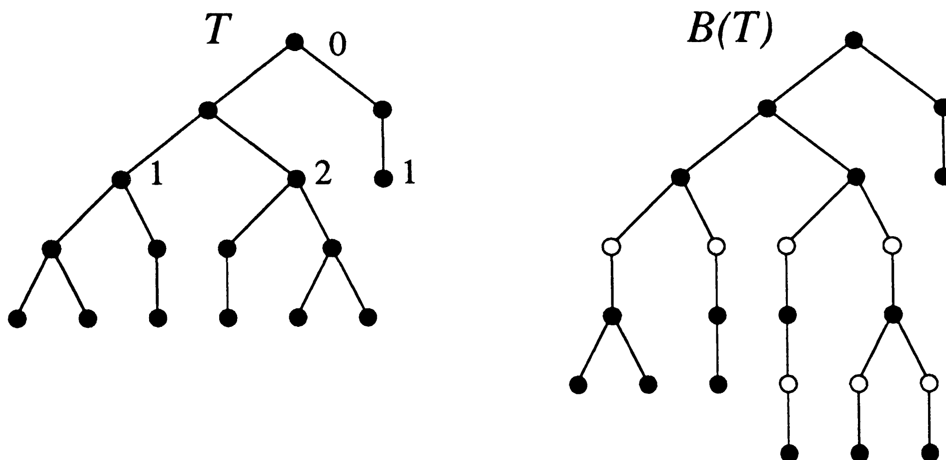


FIG. 1. *Level balancing a tree, $n = 6$. The numerical labels indicate the stretch counts chosen at those nodes. White nodes indicate dummy vertices.*

3.2. ANALYSIS OF TREE BALANCING. We now show that the n-way level-balancing transformation of § 3.1 is sufficient to evenly distribute the tree vertices among the levels in the butterfly. In particular, we show that for any tree T, no level set in $B(T)$ will contain a disproportionately large number of vertices. Since level i of the butterfly receives vertices from the ith level set of $B(T)$, this implies that tree vertices are uniformly distributed among the butterfly levels.

LEMMA 4. *For an arbitrary binary tree T, the n-way level-balancing transformation yields a tree $B(T)$ such that with high probability the total number of vertices in the* ith *level set of $B(T)$ is at most $O(M/n + 2^n)$.*

We will prove the following slightly modified (but equivalent) version. Define the ith *level set triple* of a tree to be the set of vertices from level sets i, $i + n/3$, and $i + 2n/3$. Define a partition of T into 3 zones as follows (Fig. 2). Zone 0 consists of vertices in levels kn through $kn + n/3 - 1$. Zone 1 consists of vertices in levels $kn + n/3$ through $kn + 2n/3 - 1$. Zone 2 consists of vertices in levels $kn + 2n/3$ through $(k+1)n - 1$. Each zone consists of a number of trees of maximum height $n/3$. We will show that no level

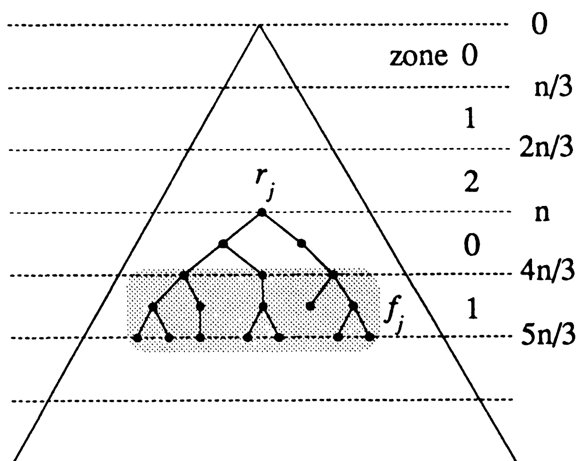


FIG. 2. *Subdivision into zones, and a forest f_j.*

set triple of $B(T)$ will receive more than $O(M/n)$ vertices from any zone of T, with high probability. Lemma 4 follows because there are only 3 zones and the number of vertices in a level set triple upper bounds the number of vertices in a level set.

The key observation is that each zone can be partitioned into a set of forests $f_1, f_2, \ldots, f_\gamma$ that contribute independently to level set triple i, for any i. We illustrate the partitioning for zone 1. Each f_j consists of all trees from zone 1 between levels $kn+n/3$ and $kn+2n/3-1$ that have a common ancestor r_j at level kn for some fixed k. Other zones are partitioned similarly.

LEMMA 5. *Let X_j denote the number of zone 1 vertices from a forest f_j placed in level set triple i of $B(T)$. Then all variables X_j are mutually independent, and $E(X_j)=3M_j/n$, where M_j is the number of vertices in f_j.*

Proof. Let the variable Y_j denote the level set triple into which the roots of the trees in f_j are placed. By definition, these roots are all placed in the level set triple given by the level set triple of r_j plus $S(r_j)$, mod $n/3$. Since the stretch counts of the r_j's are uniformly selected from $[0, n/3]$ and are mutually independent, it follows that the Y_j's are also uniformly selected from $[0, n/3]$ and are mutually independent. Since X_j is completely determined by Y_j and the stretch counts chosen at the roots of trees in f_j, it follows that the X_j are mutually independent and that $E(X_j)=3M_j/n$. □

Similarly, this lemma holds for any other zone of the tree T, except for the first section of zone 0, which contains the vertices in levels $0 \ldots n/3-1$. However, this segment of the tree contains at most $2^{n/3}-1$ nodes, which will be mapped one-to-one to nodes of the butterfly.

To prove Lemma 4 we use the following lemma, which extends a result of Hoeffding (see Lemma 9) to bound from above the probability that the sum of a set of independent random variables exceeds its mean by a given constant factor.

LEMMA 6. *Let $X_1, \ldots, X_m$ be independent random variables in the range $0, \ldots, V$ with $E[X_i]=\mu_i$. Let $X=\sum X_i$, and let $\mu=\sum \mu_i = E[X]$. Then for any constant α,*

$$\Pr[X \geqq \alpha\mu] \leqq \exp\left[-\alpha\frac{\mu}{V}\right].$$

Proof. Since the X_i's are independent, we have

$$E[e^{tX}] = \prod_i E[e^{tX_i}] = \prod_i \sum_\lambda \Pr[x_i = \lambda]\, e^{t\lambda}.$$

This expectation is maximized when only the events $[X_i=0]$ and $[X_i=V]$ have positive probability. Suppose there were some value x, not equal to 0 or V, such that $\Pr[X_i=x]=\delta>0$. Then by the convexity of e^{tX_i}, changing $\Pr[X_i=x]$ to 0 and setting $\Pr[X_i=x-1]=\Pr[X_i=x+1]=\delta/2$ would increase the expectation of e^{tX_i}. It follows that in order to maximize the expectation, the two endpoints of the interval must be the only events with positive probability. If we use Markov's inequality to put an upper bound on $\Pr[X_i=V]$, then

$$\begin{aligned} E[e^{tX}] &\leqq \prod_i \left[1-\frac{\mu_i}{V}+\left(\frac{\mu_i}{V}\right)e^{tV}\right] \\ &= \prod_i \left[1+\frac{\mu_i}{V}(e^{tV}-1)\right] \\ &\leqq \prod_i \exp\left[\frac{\mu_i}{V}(e^{tV}-1)\right] \\ &\leqq \exp\left[\frac{\mu}{V}(e^{tV}-1)\right]. \end{aligned}$$

Using Markov's inequality again, we obtain for any constant α,

$$\Pr[X > \alpha\mu] = \Pr[e^{tX} \geqq e^{t\alpha\mu}]$$
$$\leqq \frac{\exp\left[\frac{\mu}{V}(e^{tV}-1)\right]}{e^{t\alpha\mu}}.$$

This quantity is minimized at $t = \ln \alpha / V$, where

$$\Pr[X \geqq \alpha\mu] = \exp\left[-\frac{\mu}{V}(\alpha \ln \alpha - \alpha + 1)\right]$$
$$\leqq \exp\left[-\alpha\frac{\mu}{V}\right].$$

Note that in the last inequality we have weakened the result in order to achieve a simpler expression. The only important characteristic of the expression in α is that it is monotonically increasing, so that we can make the bound on the probability as small as desired by choosing a sufficiently large α. We could improve the constants in our algorithms by using the previous expression as the final result. □

Proof of Lemma 4. The X_j are independent random variables, each with mean $3M_j/n$, where M_j is the number of vertices in f_j. Clearly, no X_j can contribute more than $2^{2n/3}$ vertices, since the forest is part of a tree of height no more than $2n/3$, and the mean of $X = \sum M_j \leqq 3M/n$. Therefore, by Lemma 6, we have for any constant α

$$\Pr[X \geqq 3M/n] \leqq \exp\left[-\alpha\frac{3M}{n2^{2n/3}}\right].$$

As long as $M \geqq n^2 2^{2n/3}$, which will be the case for any $M = \Omega(N)$, this quantity is smaller than N^{-k} for some constant k, which can be made as large as desired by choosing α sufficiently large. The lemma follows. □

3.3. EFFECTIVENESS OF FLIP BITS. We now show that, given the effectiveness of the level-balancing algorithm, the flip bits suffice to distribute the tree nodes within the levels of the butterfly.

LEMMA 7. *Let W_i denote the total number of vertices in level set i in an arbitrary binary tree T. When T is grown on a butterfly with n levels, no processor from level i receives more than $O(W_i/2^n + n)$ vertices with high probability, for all i.*

In other words, whenever $W_i = \Omega(n2^n)$, each of the 2^n processors in level i will receive roughly the same number of tree vertices.

The key to the proof is the observation that the vertices placed on a processor can be attributed to a large number of mutually independent sources. To see this, partition T into subtrees $T_1, T_2, \ldots$ where each subtree is rooted at some vertex in level $kn + i$ and consists of all the descendants of that vertex between levels $kn + i + 1$ and $kn + i + n$ (see Fig. 3).

LEMMA 8. *At most one level-n vertex from each subtree T_j will be placed on any processor p on level i of the butterfly. The probability of a vertex from T_j being placed on processor p is $w_j/2^n$, where w_j denotes the number of vertices in level n of tree T_j. Furthermore, the contributions of the different subtrees to p are mutually independent.*

Proof. Any tree T_j can have at most 2^n vertices at level n, and the growth algorithm guarantees that these will be placed on distinct processors within a single level. Thus, we know that at most one vertex from a tree T_j will be placed on a given processor p in level i of the butterfly.

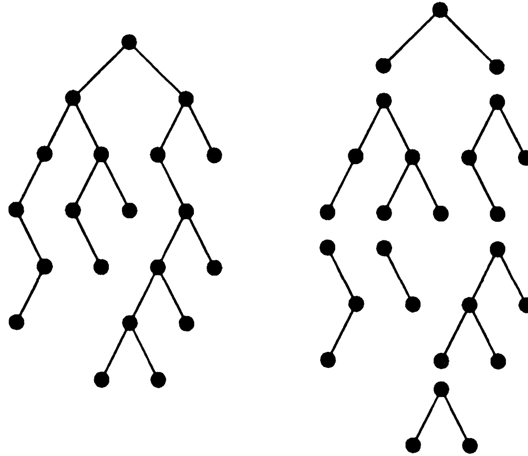


FIG. 3. *The tree T and its partition, $i=1$, $n=2$.*

It follows from the above that the number of vertices from T_j placed on p is a random variable with value either 0 or 1. The probability that any given vertex from level n of T_j will be placed on p is $1/2^n$, so the expectation of this random variable is $w_j/2^n$. Since the value of the random variable can only be 0 or 1, $w_j/2^n$ must be the probability that it is 1. Thus, the probability of a vertex from T_j being placed on p is $w_j/2^n$.

The independence between different subtrees follows because the flip bits in each subtree are picked independently. □

To complete the proof of Lemma 7, we need the following lemma (stated here without proof) from Hoeffding [8].

LEMMA 9 (Hoeffding). *If we have L independent Bernoulli trials with respective probabilities $p_1, \ldots, p_L$, with $Lp=\sum p_i$ and $m \geqq Lp+1$ an integer, the probability of at least m successess is at most $B(m, L, p)$, where $B(m, L, p) \leqq (Lpe/m)^m$.*

Proof of Lemma 7. The number of vertices placed at a processor is the sum of independent random variables corresponding to each tree T_j. The expected number of vertices is $\sum w_j/2^n = W_i/2^n$. The probability that some processor receives more than $k(W_i/2^n+n)$ vertices is at most (using Lemma 9)

$$\left(\frac{k(n+W_i/2^n)}{eW_i/2^n}\right)^{-k(n+kW_i/2^n)} \leqq (k/e)^{-kn}.$$

Thus, the probability that one of the 2^n processors in any of the n levels receives more than $k(W_i/2^n+n)$ vertices is at most

$$n2^n(k/e)^{-kn} \leqq N^{-k\log k/k_1}$$

for some constant k_1. The exponent can be made smaller than any desired constant by choosing k sufficiently large. □

THEOREM 2. *An arbitrary binary tree T with M vertices can be grown dynamically on an N processor butterfly network with dilation 2 such that with high probability the maximum load per processor is at most $O(M/N+\log N)$.*

Proof. By Lemma 4, with high probability we have $W_i=O(M/n+2^n)$ for all i, and by Lemma 7, with high probability no processor in level i will receive more than $O(W_i/2^n+n)$ vertices. Therefore, with high probability at most $O(M/N+\log N)$ vertices are mapped to any processor. □

4. An improved hypercube embedding. The butterfly can be embedded in the hypercube with dilation 2 such that each level of the butterfly is a subcube of the hypercube. Therefore, we can have the hypercube simulate any embedding algorithm for the butterfly, with a unique 2^n-node subcube simulating each level. We will take advantage of this by using a scheme that has each level (subcube) receiving at most $O(M/n+2^n)$ tree nodes and by developing a method for local distribution within these subcubes that will reduce the load on each individual processor while maintaining low congestion. We begin with some preliminaries.

4.1. EMBEDDING THE BUTTERFLY AND STAR COVERS. Let $G(x)$ be the *Grey code* value of the binary string x, defined by

$$G(x_{\log n} \dots x_1) = x_{\log n}|x_{\log n} \oplus x_{\log n-1}| \dots |x_2 \oplus x_1,$$

For any bit string x, $G(x)$ and $G((x+1) \bmod n)$ differ in exactly one bit position. For an integer i, let bin (i) be the binary representation of i. The embedding that maps butterfly processor v_l to node $G(\text{bin}\,(l))|\text{bin}\,(v)$ of the hypercube has dilation 2 and maps each level of the butterfly to a distinct 2^n-node subcube of the hypercube. Also, note that within each level l, if v and v^k differ in exactly one bit, then there is a hypercube edge between the embedded locations of the nodes $\langle l, v\rangle$ and $\langle l, v^k\rangle$.

For any node x of a 2^n-node hypercube, we define the *full star centered at* x to be the set of nodes consisting of x along with the n nodes adjacent to x. The existence of perfect one-error-correcting codes implies that when $n=2^m-1$, for some integer m there exists a collection of $2^n/(n+1)$ full stars such that every node of the hypercube belongs to precisely one star in the collection.

Suppose n is not of this form. Consider the largest n' such that $n' \leqq n$ and n' is of the form $n'=2^m-1$; thus, $n' \geqq n/2$. We can partition the hypercube into subcubes of $2^{n'}$ nodes and cover each of these with full stars. This *star cover* perfectly covers the nodes of the 2^n-node hypercube. Each of the $\Theta(2^n/n)$ stars in the star cover consists of a node x and some subset of $\Theta(n)$ (in this case, at least $n/2$) of its neighbors.

Choose a star cover for a 2^n-node hypercube, and duplicate this cover in each subcube of the $N(=n2^n)$-node hypercube that corresponds to a level of the butterfly. This collection of stars yields a star cover of the N-node hypercube; call it $\mathscr{C}$.

4.2. MODIFYING THE EMBEDDING ALGORITHM. Our discussion of the hypercube algorithm has two parts:

1. We describe a modified algorithm for embedding in the butterfly that, when simulated on a hypercube, maps at most $O(M/2^n+n)$ tree nodes to any star in the cover $\mathscr{C}$, with high probability.

2. We show how to deterministically redistribute the load within a star of the hypercube among its nodes in such a way that each node receives $O(M/N+1)$ tree nodes, the dilation remains constant, and the resulting congestion is $O(M/N+1)$.

We begin by showing how to modify the butterfly embedding algorithm given in the previous section so that when it is simulated on the hypercube, the amount of load assigned to any star in the cover $\mathscr{C}$ is balanced.

We will modify our embedding algorithm as follows. Use the embedding algorithm from § 3, but where previously we placed the children of a tree node $v \in B(T)$ that was embedded in level l into level $l+1$, choosing their locations by a random flip bit, we will now place the first child of v into level $l+2$, using a *pair* of flip bits to determine its position within the level and placing the second child (if it exists) at the location in that level determined by complementing both flip bits. It is clear that this will increase dilation by a factor of two.

Since we are embedding the level-balanced tree $B(T)$, we know that, with high probability, each level set of the tree contains $O(M/n+2^n)$ nodes. As in Lemma 5, we observe that the vertices placed in a single star come from many mutually independent sources.

Partition $B(T)$ into subtrees $T_1, T_2 \ldots$ in such a way that the root of each subtree is embedded at level $l+2$ in the butterfly (or level $l+1$ if n is odd) and each subtree contains the descendants of its root down to the nodes embedded at level l in the butterfly.

LEMMA 10. *Consider an arbitrary star S in $\mathscr{C}$, contained in the subcube simulating level l of the butterfly. Then, at most two vertices from each subtree can be placed on processors in S. Furthermore, the contributions of each subtree to S are mutually independent.*

Proof. Any subtree can have at most $2^{n/2}$ vertices placed in level l of the butterfly, and these will necessarily be placed at distinct locations within the level. Suppose that three vertices from the same subtree were mapped to the star S. Since the flip bits are chosen in pairs, any pair of these vertices must be mapped to locations that differ in an even number of bits; since they are all mapped to the same star, any pair of them must differ in exactly two bit positions. Consider the paths to each of these three vertices from their lowest common ancestor; call this vertex x. Clearly, two of the vertices must be descendants of one child of x, and one must be a descendant of the other. The vertex (call it y) that is the lone descendant of one of the children of x now differs from both of the other two vertices in two bit positions that are not corrected elsewhere in the tree. However, at some point the paths of the other two vertices diverge (since they are placed on different processors in level l), and y's path cannot duplicate the flip bits on both paths simultaneously. Therefore, y differs from one of the other two vertices in at least four bit positions, contradicting the supposition that all three vertices were in the same star in level l. Therefore, at most two vertices from the same subtree can be placed in the star S.

The independence between different subtrees follows from the fact that the flip bits are picked independently in each subtree. □

LEMMA 11. *We can embed an arbitrary binary tree T with M nodes into an N-node hypercube such that, with high probability, no star in the cover $\mathscr{C}$ receives more than $O(M/2^n+n)$ tree nodes.*

Proof. Consider an arbitrary star S in level l from the cover $\mathscr{C}$. Let X_i be the number of tree nodes from subtree T_i that are assigned to processors in S. The X_i are independent random variables, each with maximum value 2 (from Lemma 10) and mean $\Theta(m_i n/2^n)$, where m_i is the number of leaves of the subtree T_i. It follows that the mean of $X=\sum X_i$ is $\Theta(W_l n/2^n)$, where W_l is the size of level set l of the tree. Since we are balancing levels by embedding the tree $B(T)$, we have with high probability $W_l=O(M/n+2^n)$, so that the mean of X is less than $c(M/2^n+n)$ for some constant $c>0$. Thus, as we did in the proof of Lemma 4, we can apply Lemma 6 and conclude that with high probability no star in $\mathscr{C}$ receives more than $O(M/2^n+n)$ tree nodes. □

4.3. REDISTRIBUTING LOAD WITHIN STARS. With high probability, each star in the cover has at most $O(M/2^n+n)$ tree nodes assigned to its $\Theta(n)$ nodes. From this point on, we will assume that this is the case, and we would like to redistribute the $O(M/2^n+n)$ load on each star evenly among the $\Theta(n)$ nodes of the star, using the hypercube edges connecting butterfly nodes within the same level, so that two conditions hold:

1. Each node gets at most $O(M/N+1)$ load.

2. We can choose paths of constant length between the redistributed locations of adjacent tree nodes so that the congestion on any hypercube edge is at most $O(M/N+1)$.

If these two conditions can be achieved by a redistribution scheme that runs dynamically as the tree is embedded, then we have an embedding algorithm that, with high probability, achieves load $O(M/N+1)$, dilation $O(1)$, and congestion $O(M/N+1)$—simultaneously optimizing load and dilation to within constant factors. In addition, the congestion will be optimal if $M=O(N)$.

Place an $O(M/N+1)$ upper limit (with the choice of constant depending on the constant in Lemma 11 and the number of elements in each star) on the number of tree nodes that can be assigned to a single node. All additional load is sent to some other node in the star that has room. It is clear that we have sufficient capacity over each star to handle the load and that we can maintain constant dilation, since any pair of nodes in the star are at a distance of at most two from each other. In addition, we will have maximum load $O(M/N+1)$ at each node of the hypercube. Note that this method is not the same as allowing process migration—each tree node is redistributed *before* it is embedded into the hypercube. Once the node's redistributed location is determined, it is embedded there permanently.

To keep the congestion low, we must choose the path between the final (redistributed) locations of adjacent tree nodes u and v carefully. The first step is to choose paths from each node's original embedded location to its redistributed location. Clearly, if a node being redistributed from v^i to v^j in the star centered at v were always to choose the path $v^i \to v \to v^j$, then the congestion along some of the edges of the star might be as high as the load of the original embedding—$O(M/2^n+n)$. Instead, when we redistribute one tree node from node v^i to node v^j in the star centered at v in the hypercube (load coming from or going to the center is redistributed directly), we will choose the path $v^i \to v^{ij} \to v^j$ rather than through the center of the star.

LEMMA 12. *If all nodes being redistributed among points of the star centered at v choose paths of the form $v^i \to v^{ij} \to v^j$ rather than paths through the center of the star, then the resulting congestion on edges between nodes in a single butterfly level due to this redistribution is $O(M/N+1)$.*

Proof. For each star in the cover, consider the corresponding *extended star*, which consists of the star centered at v plus all vertices v^{ij} such that both v^i and v^j are in the star. Then the edges in the extended star consist precisely of those paths along which load can be redistributed in the star centered at v, and each of these edges is used by only one path between a pair of nodes in the star. Therefore, since each node in the star ends up with load $O(M/N+1)$, the total congestion on each edge due to redistribution within that star is also $O(M/N+1)$. All that remains is to observe that any edge in the hypercube is in at most two extended stars. Thus, the total congestion it receives from redistribution is $O(M/N+1)$. □

We are still not quite finished, however. We must demonstrate how to select a path in the hypercube between the redistributed locations of adjacent tree nodes u and v. If we were to choose the simplest path between their redistributed locations, moving first from the redistributed location of u to its original location and then to the original location of v and on to its redistributed location, the congestion of the embedding along the edges of the butterfly could still be as high as the load of the embedding *before* redistribution—$O(M/2^n+n)$. To reduce the congestion of $O(M/N+1)$, we must choose the paths more carefully.

Let l be the level of the butterfly to which u is mapped; then v is mapped to level $l+2$. Furthermore, their positions within their respective butterfly levels differ in at

most two bit positions (before the redistribution just described). We consider here the case in which both u and v are both initially mapped and redistributed to some point of a star rather than the center. When one or both of them are mapped or redistributed to the center of a star, the argument is even simpler.

Let x and y be the centers of the stars to which u and v, respectively, are mapped. Let p and q be the dimensions within the star to which u is mapped and redistributed, and likewise r and s for v. Let f_1, f_2 be the flip bits selected when v is embedded as a child of u. We then define the path from u, which is redistributed from x^p to x^q in level l, and v, which is redistributed from y^r to y^s in level $l+2$, as follows (this procedure is illustrated in Fig. 4):

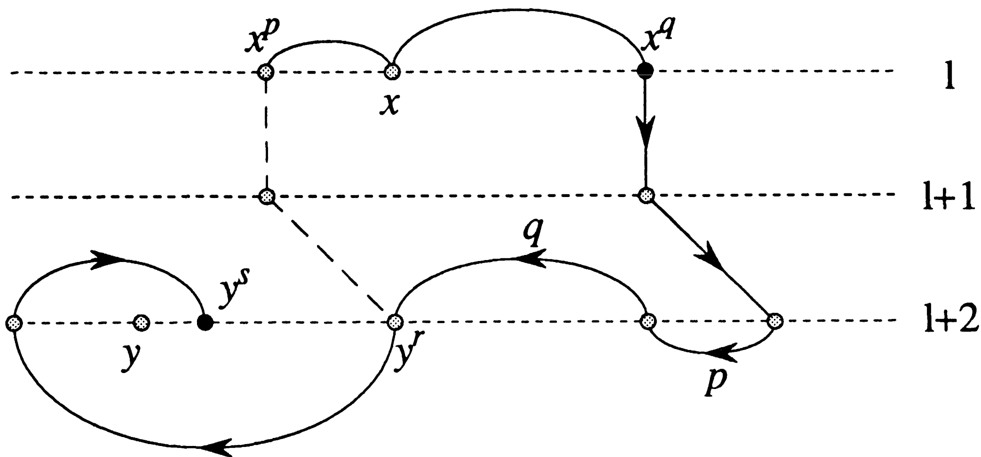


FIG. 4. *The path chosen between redistributed node locations. The dashed lines indicate the path determined by the flip bits, before redistribution. The first pair of directed edges also shows this choice of flip bits. The second pair undoes the redistribution at level l. The last pair balances the load at level $l+2$.*

1. Move from level l to level $l+1$ to level $l+2$ along the edges determined by the flip bits f_1, f_2.
2. Flip the bits in positions p, then q, in effect undoing the redistribution of u that was performed in level l. We are now at y^r, the original location of v before redistribution.
3. Flip the bits in positions s, then r. This takes us to y^s, the redistributed location of v in its star in level $l+2$.

To show that the congestion is $O(M/N+1)$ in this case, it suffices to show two things: first, that the congestion along each edge of the butterfly from paths between levels is $O(M/N+1)$; second, that the congestion along each hypercube edge connecting nodes within a butterfly level due to redistribution is $O(M/N+1)$.

Consider an arbitrary butterfly edge. There are at most two nodes of the butterfly that, when choosing the paths to their descendants, can use that edge. Since after redistribution each of these nodes has load $O(M/N+1)$ (this is the reason why we start the paths at the redistributed locations), the congestion along the edge being considered can also be at most $O(M/N+1)$.

The congestion on hypercube edges connecting butterfly nodes within a level has two sources: (1) the redistribution of the nodes embedded to that level, and (2) undoing the redistribution of the parents of the nodes embedded to that level.

It is immediate from Lemma 12 that the total congestion from the first source does not exceed $O(M/N+1)$. We can partition the congestion derived from the second source into four sets according to the flip bits chosen along the paths from the parents of the nodes in the level we are considering. Each fixed setting of flip bits determines a bijective map of the nodes and , therefore, of the hypercube edges between butterfly nodes in the level, from two levels above to the current level. The congestion on any edge from undoing the redistribution of parents equals the congestion on its preimage from the original redistribution. The congestion derived from each of the four sets is therefore $O(M/N+1)$, so the total congestion derived from undoing the redistributions is also $O(M/N+1)$. It follows that the entire congestion on any edge is $O(M/N+1)$.

The existence of this algorithm suffices to prove the following theorem.

THEOREM 3. *An arbitrary binary tree T with M vertices can be dynamically grown on an N processor hypercube with constant dilation such that with high probability the maximum load is $O(M/N+1)$ and the congestion is $O(M/N+1)$.*

5. A lower bound for deterministic algorithms. In this section we prove that any deterministic algorithm for dynamically embedding an M-node binary tree in an N-node hypercube ($M \geqq N$) that maintains maximum load cM/N must have not only maximum but also *average* dilation $\Omega(\sqrt{\log N}/c^2)$. It follows that any deterministic embedding algorithm that achieves $O(M/N+1)$ load must necessarily result in embeddings with dilation $\Omega(\sqrt{\log N})$ for some binary trees. Thus, any embedding algorithm that simultaneously optimizes maximum load and dilation to within constant factors must be randomized.

THEOREM 4. *Any deterministic algorithm for dynamically embedding binary trees in an N-node hypercube that achieves load cM/N for a tree with M ($\geqq N$) nodes must have average edge length $\Omega(\sqrt{\log N}/c^2)$.*

Proof. Let cM/N be the load maintained by the embedding algorithm when embedding an M-node binary tree. Define the *size* of a node in the hypercube to be the number of 1's in the n-bit string associated with the node. Partition the hypercube into $6c$ blocks, each block corresponding to some range of node sizes and containing $N/6c$ nodes. Since there are at most $O(N/\sqrt{\log N})$ nodes of any size, each block must contain at least $\Omega(\sqrt{\log N}/c)$ different sizes. This means that any two nodes that are in nonadjacent blocks are at distance $\Omega(\sqrt{\log N}/c)$ from each other.

Choose an arbitrary $M \geqq N$, and grow a path of $M/2$ nodes, starting at the root. At this point, some block must contain $M/12c$ tree nodes; choose such a block. We will continue growing the tree from the $M/12c$ nodes in the chosen block. Grow paths from each of these tree nodes simultaneously, stopping each path's growth when it reaches a hypercube node that is neither in the chosen block nor in a block adjacent to it. The total number of nodes in the chosen block and adjacent blocks is at most $N/2c$; since the algorithm maintains load cM/N, this set of nodes contains at most $(cM/N)(N/2c)=M/2$ tree nodes. It follows that the total length of the $M/12c$ paths grown is at most $M/2$, so the tree being considered has at most M nodes.

Now we can calculate the average edge length. Since each of the $M/12c$ paths connects a node in the chosen block to a node in some nonadjacent block, the total edge length in these paths is at least $(M/12c)\cdot\Omega(\sqrt{\log N}/c)=\Omega(M\sqrt{\log N}/c^2)$. Since the entire tree contains at most M edges, it follows that the average edge length of the embedding is $\Omega(\sqrt{\log N}/c^2)$. □

6. Remarks. The embedding in § 4 achieves dilation at most 12, assuming that we do not specifically embed the nodes that are inserted as part of our level-balancing transformation but rather connect their parents' locations directly to their childrens'.

One edge of T corresponds to at most 2 edges of $B(T)$, each of which corresponds to 2 butterfly edges. In the embedding of the butterfly into the hypercube, each butterfly edge corresponds to 2 edges of the hypercube. The redistribution algorithm adds at most 4 edges to the resulting path, for a total of 12 hypercube edges. We can easily reduce this to 10 without increasing the load or congestion by modifying the level-balancing scheme from § 3 and the strategy for selecting flip bits in § 4, and we expect that there is room for further improvement.

Natural extensions of the embedding algorithms described in this paper work for binary trees that can grow and shrink from the top as well as from the bottom. Formally, this means that as well as allowing any leaf to delete itself and any node with fewer than two children to spawn another child, we (1) allow the root of the tree to spawn a new parent, with the old root as its only child, and (2) allow the root of the tree to delete itself if it has only one child—this child then becomes the new root of the tree. Minor changes in the arguments in §§ 2, 3, and 4 suffice to show that these algorithms perform just as well on this problem as the original algorithms did for the case of growth and shrinkage only at the leaves. We also expect that our techniques can be made to work for arbitrary trees of small degree and that they may prove useful for finding embeddings in other networks, such as the shuffle-exchange graph.

7. Acknowledgments. We would like to thank Umesh Vazirani for suggesting the level-balancing transformation and John Hartman for carrying out simulations that bolstered our initial intuitions.

REFERENCES

[1] S. N. BHATT AND J.-Y. CAI, *Take a walk, grow a tree*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 469–478.

[2] S. N. BHATT, F. R. K. CHUNG, J.-W. HONG, F. T. LEIGHTON, AND A. L. ROSENBERG, *Optimal simulations by butterfly networks*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 192–204.

[3] S. N. BHATT, F. R. K CHUNG, F. T. LEIGHTON, AND A. L. ROSENBERG, *Optimal simulations of tree machines*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1986, pp. 274–282.

[4] S. N. BHATT AND I. IPSEN, *How to embed trees in hypercubes*, Tech. Report RR-443, Department of Computer Science, Yale University, New Haven, CT, 1988.

[5] M. Y. CHAN, *Dilation-2 embeddings of grids into hypercubes*, Tech. Report UTDCS 1-88, Computer Science Department, University of Texas at Dallas, Dallas, TX, 1988.

[6] D. S. GREENBERG, L. S. HEATH, AND A. L. ROSENBERG, *Optimal embeddings of butterfly-like graphs in the hypercube*, Math. Systems Theory, 23 (1990), pp. 61–77.

[7] C. T. HO AND S. L. JOHNSSON, *Embedding generalized pyramids in hypercubes*, Tech. Report, Department of Computer Science, Yale University, New Haven, CT, 1988, in preparation.

[8] W. HOEFFDING, *On the distribution of the number of successes in independent trials*, Ann. Math. Statist. 27 (1956), pp. 713–721.

[9] R. M. KARP AND Y. ZHANG, *A randomized parallel branch-and-bound procedure*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 290–300.

[10] R. KOCH, T. LEIGHTON, B. MAGGS, S. RAO, AND A. ROSENBERG, *Work-preserving emulations of fixed-connection networks*, in Proc. 21st Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 227–240.

SIAM J. COMPUT.
Vol. 21, No. 4, pp. 655-670, August 1992
© 1992 Society for Industrial and Applied Mathematics
003

CIRCUIT DEFINITIONS OF NONDETERMINISTIC COMPLEXITY CLASSES*

H. VENKATESWARAN†

Abstract. This paper considers restrictions on Boolean circuits and uses them to obtain new uniform circuit characterizations of nondeterministic space and time classes. It also obtains characterizations of counting classes based on nondeterministic time bounded computations on the arithmetic circuit model. It is shown how the notion of semi-unboundedness unifies the definitions of many natural complexity classes.

Key words. Boolean circuits, arithmetic circuits, NP, P, skew circuits, semi-unboundedness

AMS(MOS) subject classifications. 68Q05, 68Q10, 68Q15, 94C99

1. Introduction. Uniform Boolean circuits have provided a very useful framework to study some of the important issues that arise in Turing machine based complexity theory. Close connections have been established between complexity classes based on uniform circuits and those based on the machine model [2], [5], [6], [11], [12], [14]. In one direction, complexity classes defined by the circuit model have been characterized by the machine model. NC is a well-known example of such a complexity class; it was defined by the uniform Boolean circuit model [11] and has been characterized by using the alternating Turing machine model by Ruzzo [14]. In the other direction, traditional complexity classes based on the machine model have been characterized by the circuit model. The definition of the class P using Boolean circuits [8], [12] is probably the first such result. Other results of this nature are the characterizations in the circuit model of the classes AC^1 [16] and LOGCFL [18]. The results by Ruzzo [14] also make it possible to obtain circuit characterizations of complexity classes defined by using alternating Turing machines. The work reported here extends these results to characterize classes defined by nondeterministic Turing machines.

1.1. OVERVIEW. In the first part of this paper, we consider restrictions of Boolean circuits and use them to characterize nondeterministic space and time classes. This includes a characterization of nondeterministic time classes on the semi-unbounded fan-in circuit model. Semi-unbounded fan-in circuits, which are Boolean circuits in which the OR gates are allowed arbitrary fan-in and the AND gates have bounded fan-in, have been previously used to define the class LOGCFL [18]. We define skew circuits as Boolean circuits in which all but one input of every AND gate are circuit inputs and use them to characterize nondeterministic space and time classes. Nondeterministic space is defined in terms of the size of such circuits, and nondeterministic time is shown to correspond to the depth of these circuits. This should be contrasted with the well-known correspondences between deterministic time and Boolean circuit size [12] and between nondeterministic space and Boolean circuit depth [2].

In the second part of the paper, we use the monotone arithmetic circuit model to characterize counting classes based on nondeterministic time bounded computations. Monotone arithmetic circuits are arithmetic circuits over the domain of nonnegative

* Received by the editors June 26, 1989; accepted for publication (in revised form) July 22, 1991. This work was supported by the National Science Foundation under grant CCR-8711749. A preliminary version of this paper was presented at the eighth annual conference on Foundations of Software Technology and Theoretical Computer Science held at Pune, India, December 21-23, 1988. A preliminary version also appeared as Georgia Institute of Technology Technical Report GIT-ICS-88-09.

† College of Computing, Georgia Institute of Technology, Atlanta, Georgia 30332-0280 (venkat@cc.gatech.edu).

integers that use only the addition and multiplication operations. An interesting consequence of this characterization is the definition of the well-known counting class $\#P$ [17] as the set of functions computed by uniform families of monotone arithmetic circuits that have polynomial depth and polynomial degree. The degree measure here refers to the algebraic degree of the polynomial associated with the circuit.

It would be appropriate to mention some interesting consequences of the characterization results presented here.

- The circuit characterizations of NP presented here are, to our knowledge, the first uniform circuit characterizations of this important complexity class. Of particular interest is the definition of NP as the class of languages accepted by uniform families of semi-unbounded fan-in circuits of exponential size and log depth. This provides a framework to study some interesting questions about the class NP. Recently, Borodin et al. [3] proved that if a language is accepted by a family of semi-unbounded fan-in circuits of size $Z(n)$ and depth $D(n)$, then its complement is accepted by a family of semi-unbounded fan-in circuits of size polynomial in $Z(n)$ and depth $O(D(n)+\log Z(n))$. Their result does not apply directly to NP, since it only shows that CO-NP is accepted by semi-unbounded fan-in circuits of exponential size and polynomial depth. The relevant question here is whether the classes accepted by size $Z(n)$ and depth $o(\log Z(n))$ semi-unbounded fan-in circuits are closed under complement. It is known that the classes accepted by polynomial size and $o(\log n)$ depth semi-unbounded fan-in circuits are not closed under complement [18]. Another complexity question pertaining to NP that can be phrased in this model is its relationship with the other classes definable using semi-unbounded fan-in circuits. A candidate class for comparison is the class LOGCFL. It is known that LOGCFL can be characterized as the class of languages accepted by uniform families of polynomial size and log depth semi-unbounded fan-in circuits [18]. Therefore, the separation between NP and LOGCFL now becomes a question of the relative power of exponential size and polynomial size semi-unbounded fan-in circuits of logarithmic depth.
- The skew Boolean circuits provide a model to rephrase many of the famous separation questions among complexity classes. Thus the relationship between P and NLOG translates into the question of the relative power of polynomial size Boolean circuits and polynomial size skew Boolean circuits. The P versus PSPACE question becomes one of comparing the relative power of polynomial size Boolean circuits and exponential size skew Boolean circuits. As another interesting example, the NP versus PSPACE question can be phrased as the question about polynomial depth for skew Boolean circuits versus polynomial depth for general Boolean circuits.
- The arithmetic characterization of $\#P$ presented here is the first alternative characterization of this class. It enables us to rephrase the famous open question about the relationship between $\#P$ and NP in terms of the relative power of arithmetic and Boolean circuits. It also touches on the power of arithmetic circuits over monotone arithmetic circuits.
- These characterizations also make it possible to identify appropriate circuit value problems that are complete for each of these complexity classes.
- The semi-unbounded fan-in circuit model seems useful to capture the definitions of many nondeterministic complexity classes (see Table 1).

This paper is organized as follows. Section 1.2 contains some preliminary definitions. Boolean circuit characterizations of nondeterministic space and time classes are in § 2. Some characterizations of nondeterministic time that follow as simple consequences of known results are presented in § 3. A monotone arithmetic circuit characterization of counting classes based on nondeterministic time is presented in § 4.

1.2. PRELIMINARIES.

Boolean circuits. A *Boolean circuit* G_n with n inputs is a finite acyclic directed graph with vertices having indegree zero or at least two and labeled as follows. Vertices of indegree zero are labeled from the set $\{0, 1, x_1, x_2, \cdots, x_n, \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n\}$. All other vertices (also called *gates*) are labeled either AND or OR. It should be noted that not including negation gates in the definition of a Boolean circuit is done with no loss of generality [7]. Vertices with outdegree zero are called *outputs.* The evaluation of G_n on inputs of length n is defined in the standard way. Typically, only circuits with one output vertex will be considered. This makes it convenient to consider circuits as language acceptors.

The *size* $C(G_n)$ of a circuit G_n is the number of edges in G_n. The *depth* of a vertex v in a circuit is the length of a longest path from any input to v. The depth of a circuit is the depth of its output vertex.

The language L_n accepted by a Boolean circuit G_n is the set of all length n strings on which G_n evaluates to one. A *family* of circuits is a sequence $\{G_n \mid n = 0, 1, 2, \cdots\}$, where the nth circuit G_n has n inputs. The language L accepted by a family $\{G_n\}$ of circuits is defined as follows: $L = \bigcup_{n \geqq 0} L_n$, where L_n is the language accepted by the nth member G_n of the family.

Skew Boolean circuits. Let G be a Boolean circuit. An AND gate v in G is said to be a *skew gate* if it has at most one input that is not an input of G. Without loss of generality, we will assume that all but one of its inputs are inputs to the circuit G. We will refer to the input of v that is not an input to G as a *nonskew* input of v. The circuit G is said to be a *skew circuit* if all AND gates in it are skew gates. A family $\{G_n\}$ of Boolean circuits is said to be a *skew circuit family* if all its members are skew circuits.

Note: One can define skewness with respect to OR gates also, but we will not pursue that in this paper.

Semi-unbounded fan-in Boolean circuits. A family of Boolean circuits is said to have semi-unbounded fan-in if there exists a constant $c > 0$ such that for any circuit in the family, the OR gates in the circuit can have unbounded fan-in and all the AND gates have fan-in at most c.

Semi-unbounded alternating Turing machines. An alternating Turing machine is *semi-unbounded* if there are no two consecutive universal configurations along any path in the computation tree of the machine. Without loss of generality, we will assume that every universal configuration of a semi-unbounded alternating Turing machine has exactly two existential configurations as immediate successors.

Uniformity. We will use the following notion of uniformity, called *U_D-uniformity,* defined by Ruzzo [14]. Define the *direct connection language L_{DC}* of a family of Boolean circuits to be the set of strings of the form $\langle n, g, y\rangle$ such that either (i) g and y are gate names and y is an input of the gate g, or (ii) g is a gate name and y is the type of the gate g; that is, y is one of AND or OR or an input to G_n or its negation. A family $\{G_n\}$ of Boolean circuits of size $C(n)$ is said to be *uniform* if the corresponding direct connection language can be recognized by a deterministic Turing machine in time $O(\log C(n))$.

For the space characterization results in § 2, it would have been sufficient to consider *log-space uniformity,* defined by Borodin and by Cook [4]. However, a stronger uniformity condition is needed for the time characterization results to avoid the possibility of having a uniformity machine that is more powerful than the class being characterized. Such would be the case, for instance, in Theorem 7 if we had used log-space uniformity, since NTIME $(T(n)) \subseteq$ DSPACE $(T^{O(1)}(n))$.

Accepting subtrees [19]. The notion of an accepting subtree of a Boolean circuit given an input on which it evaluates to one is analogous to the notion of accepting subtrees of machines.

Let B be a Boolean circuit, and let $T(B)$ be its tree equivalent. (The tree equivalent of a graph is obtained by replicating vertices whose outdegree is greater than one until the resulting graph is a tree.) Let x be an input on which B evaluates to one. An *accepting subtree* H of the circuit B on input x is a subtree of $T(B)$ defined as follows:

- H includes the output gate,
- for any AND gate v included in H, all the immediate predecessors of v in $T(B)$ are included as *its* immediate predecessors in H,
- for any OR gate v included in H, exactly one immediate predecessor of v in $T(B)$ is included as *its* only immediate predecessor in H, and
- any input vertex of $T(B)$ included in H has value one as determined by the input x.

It is easy to verify the fact that the circuit B evaluates to one given the input x if and only if there is an accepting subtree of $T(B)$ on input x.

Tree size [19]. The tree size measure for Boolean circuits can now be defined analogous to the tree size measure for alternating Turing machines [14].

The circuit B_n is said to have tree size $Z(n)$ if, for every input x accepted by B_n, there exists an accepting subtree with at most $Z(n)$ vertices.

Degree [15]. We define the *degree* of a circuit to be the algebraic degree of the polynomial computed by the circuit. Thus, the constants have degree zero, the circuit inputs have degree one, the degree of an OR vertex is the maximum of the degrees of its inputs, and the degree of an AND vertex is the sum of the degrees of its inputs.

The following lemma [18] establishes a relationship between the degree and tree size measures for Boolean circuits.

LEMMA 1. *Let $D(n)$, $Z(n)$, and $d(n)$ be the degree, tree size, and depth, respectively, of a Boolean circuit B_n. Then,*

$$Z(n) \leqq D(n)d(n).$$

Proof. The results to be proved also holds when the Boolean circuits considered have unbounded fan-in. Let x be an input accepted by the circuit B_n. By hypothesis, there is an accepting subtree H of B_n of size at most $Z(n)$. Let v be any vertex in H. Then the lemma follows from the claim below. The claim itself is proved by induction on depth.

Claim. Let $Z(v)$ be the number of vertices in the subtree of H rooted at v, $D(v)$ be the degree of v, and $d(v)$ be the depth of v. Then,

$$Z(v) \leqq D(v)d(v). \qquad \square$$

2. Characterizations of space and time classes. This section contains the characterizations of nondeterministic space and time classes in terms of skew circuits and semi-unbounded fan-in circuits. Theorem 6 relates simultaneous space and time bounded nondeterministic classes to simultaneous size and depth bounded skew circuits. In this respect, it is similar to the result of Ruzzo [14] relating simultaneous space and time bounded alternating classes to simultaneous size and depth bounded circuits. However, the correspondence between the time and depth bounds in Theorem 6 is only within a polynomial, as opposed to the correspondence within a constant factor between circuit depth and alternating time shown by Ruzzo [14].

In the proof of Lemma 3 below, we choose to use the alternating Turing machine model instead of directly constructing a semi-unbounded fan-in circuit corresponding

to a skew circuit. This is done to simplify the proof by using known simulation techniques. It also provides a new characterization of nondeterministic time on the alternating Turing machine model (see Theorem 9). The correspondence between the machine and circuit models will be established through a sequence of lemmas.

LEMMA 2. *For* $S(n)=\Omega(\log n)$, $T(n)=\Omega(n)$, *and* $S(n)\leqq T(n)$,

NSPACE,TIME $(S(n), T(n))\subseteq$ Uniform Skew Circuit SIZE,DEPTH $(2^{O(S(n))}, T(n))$.

Proof. Let L be accepted by a nondeterministic Turing machine M in $S(n)$ space and $T(n)$ time. The construction of a circuit family $\{G_n\}$ that accepts the same language as M can be accomplished by standard techniques [14], [18]. For the sake of completeness, we will outline below the construction of G_n, the nth member of this family.

The configurations of M can be classified into two types: existential and read. We will assume that M is deterministic while reading inputs.

For $0\leqq t\leqq T(n)$ and a configuration c of M using space $S(n)$, there is a gate in the circuit in one of the following forms: $[t, c]$ or $[t, c, i]$ or $[t, c, i, b]$, where $0\leqq i\leqq n$ is an integer and b is either zero or one. The first component t in a gate name is used to avoid cycles in the circuit. The type of a gate of the form $[t, c]$ ($[t, c, i], [t, c, i, b]$) is OR (OR, AND, respectively).

Let c_I be the initial configuration of M. The output gate is $[0, c_I]$.

The inputs of a gate are constructed as follows. Consider a gate $[t, c]$ corresponding to a nonread configuration c of the machine. If $t+1>T(n)$, it has only one input, namely, the constant zero. Otherwise, its inputs are constructed from the set D of all configurations reachable by M in one move from c. There will be one input corresponding to each $d\in D$. For any $d\in D$, if d uses space $>S(n)$, then the corresponding input is the constant zero. For all other $d\in D$, there are two cases. If d is an existential configuration, the corresponding input is the gate $[t+1, d]$ and its inputs are constructed recursively. If d is a read configuration in which M reads the ith symbol, the corresponding input is an OR gate $[t+1, d, i]$ with two inputs: $[t+1, d, i, 0]$ and $[t+1, d, i, 1]$. The gate $[t+1, d, i, 0]$ is an AND gate with two inputs: NOT x_i, where x_i is the ith input, and the gate $[t+2, e]$, where e is the configuration to which M moves from the read configuration d if the ith input read has value zero. The inputs of the gate $[t+2, e]$ are constructed recursively. The gate $[t+1, d, i, 1]$ is constructed in an analogous fashion.

It is clear from the construction of G_n above that it is a skew circuit. The only AND gates constructed correspond to the read configurations of M. It is easy to show that $\{G_n\}$ accepts the same language as M. The size of the resulting circuit is $2^{O(S(n))}$. Its depth is $T(n)$.

It can be verified that the direct connection language of $\{G_n\}$ can be recognized by a deterministic Turing machine using $O(S(n))$ time, thus showing that the circuit family $\{G_n\}$ is uniform □

LEMMA 3. *For* $S(n)=\Omega(\log n)$, $T(n)=\Omega(n)$, *and* $S(n)\leqq T(n)$,

Uniform Skew Circuit SIZE,DEPTH $(2^{O(S(n))}, T(n))$

$\subseteq$ Uniform Semi-Unbounded Fan-in Circuit SIZE,DEPTH $(2^{O(S(n))}, \log T(n))$.

Proof. Let $\{G_n\}$ be a uniform family of skew circuits with the given size and depth bounds. Then $\{G_n\}$ has tree size polynomial in $T(n)$. An alternating Turing machine M that simulates G_n on an input x of length n can be constructed as in the simulation by Ruzzo [13] of a space and tree size bounded alternating Turing machine by a space and time bounded alternating Turing machine. The machine M is semi-bounded and uses space $O(S(n))$, alternations $O(\log T(n))$, and time $T^{O(1)}(n)$. Let the time used by M be $T'(n)=T^a(n)$ for some constant $a\geqq 1$. Furthermore, M is in a normal form

such that only one input symbol is read along any path of the machine's computation tree. A uniform family $\{H_n\}$ of semi-unbounded fan-in circuits, with size $2^{O(S(n))}$ and depth $O(\log T(n))$, that accepts the same language as M can be constructed by adapting known techniques [18]. The basic idea of the construction is to make as inputs to an OR (AND) gate all nonexistential (nonuniversal) configurations of M reachable through only existential (universal) configurations.

We will outline the construction of the nth member H_n of this family. The configurations of M are assumed to be one of the following three types: existential, universal, and read.

Let $D(n) = \lceil \log_2 T'(n) \rceil$.

Gates in the circuit H_n are all of the form $[c]$ or $[d']$ or $[c, d]$ or $[s, c, d]$ or $[s, c, d, e]$, where $0 \leqq s \leqq D(n)$ and c, d, and e are all configurations of M. The output gate of H_n is $[r_0]$, where r_0 is the initial configuration of M. In general, the type of a gate of the form $[c]$ is OR (AND) if the type of the configuration c is existential (respectively, universal). Given a gate $[c]$, its inputs are defined as follows.

Case 1. $[c]$ is an OR gate. Its inputs are gates $[c, d]$ for all configurations d that are not existential. Each of the gates $[c, d]$ is an AND gate and it has two inputs $[0, c, d]$ and $[d']$ defined as follows.

- The gate $[0, c, d]$ is the output of a $D(n)$ depth semi-unbounded fan-in circuit that checks that in M the configuration d is reachable from the configuration c by using only existential configurations of M. The following is a description of such a reachability circuit [18].

Given a gate $[s, c, d]$ with $0 \leqq s \leqq D(n)$, the goal is to describe a subcircuit of which this gate is the output, such that the subcircuit checks that c is reachable from d in G_n by using a path of at most $2^{D(n)-s}$ OR gates (see also the construction by Borodin [2]).

If d is an immediate predecessor of c in G_n, then $[s, c, d]$ is the constant one. Otherwise, if $s+1 > D(n)$, then $[s, c, d]$ is the constant zero. Otherwise, the gate $[s, c, d]$ is an OR gate. Its inputs are gates $[s+1, c, d, e]$ for all OR gates e in G_n. Each of the gates $[s+1, c, d, e]$ is an AND gate, and it has the two inputs $[s+1, c, e]$ and $[s+1, e, d]$. These two subcircuits are constructed recursively.

- The gate $[d']$ is an OR gate with a single input $[d]$ defined as follows. Suppose d is a read configuration with a, i on its index tape. Then $[d]$ is the ith input to H_n if $a = 1$, and $[d]$ is the complement of the ith input to H_n if $a = 0$. If d is not a read configuration, then $[d]$ is an AND gate. Its inputs are constructed recursively.

Case 2. $[c]$ is an AND gate. Let d_1, d_2 be the existential configurations of M that immediately succeed the configuration c. The inputs to $[c]$ are the OR gates $[d_1]$ and $[d_2]$. The inputs to these two OR gates are constructed recursively.

The circuit H_n has size $2^{O(S(n))}$ and depth $O(\log T(n))$. Note that the OR gates in H_n may have exponential fan-in, whereas the fan-in of the AND gates is bounded by a constant. It is easy to show that G_n and H_n accept the same language. It is also straightforward to check that the direct connection language for the circuit family $\{H_n\}$ can be recognized by a deterministic Turing machine in time $O(S(n))$. □

LEMMA 4. *For* $S(n) = \Omega(\log n)$, $T(n) = \Omega(n)$, *and* $S(n) \leq T(n)$,

$$\text{Uniform Semi-Unbounded Fan-in Circuit SIZE,DEPTH } (2^{O(S(n))}, \log T(n)) \subseteq \text{NSPACE,TIME } (S(n) \log T(n), T^{O(1)}(n)).$$

Proof. This follows from the simulation of semi-unbounded fan-in circuits by nondeterministic auxiliary pushdown automata by Venkateswaran [18]. In this case, we are interested in the space and time used in the simulation.

Let L be accepted by $\{G_n\}$, a uniform family of semi-unbounded fan-in circuits with size $2^{O(S(n))}$ and depth $O(\log T(n))$. Given x of length n, a nondeterministic machine M checks whether the circuit evaluates to one on x by doing a depth-first evaluation. The machine M maintains a stack to do the circuit evaluation.

M begins the simulation with the output gate r_0. Given a gate v and its type, M checks that v evaluates to one on x as follows. Let $C(v)$ denote the configuration of M as it begins checking the gate v.

Case 1. v is an OR gate. M existentially guesses one of its true inputs u and its type and verifies with the uniformity machine that the guesses are correct. It then recursively checks that the gate u evaluates to one.

Case 2. v is an AND gate. Then it has a constant number, say k, inputs. M existentially guesses these inputs, say, $v_1, \cdots, v_k$, and their types and verifies with the uniformity machine that the guesses are correct. M then pushes the gates $v_2, \cdots, v_k$ onto the stack. A gate's type is also pushed onto the stack with the gate. M then recursively checks that v_1 evaluates to one.

Case 3. v is an input to the circuit. If its value is zero, M rejects. Suppose v has value one. M makes its final pop move and accepts if the stack is empty. Otherwise, M pops a gate u and its type from the stack and recursively checks that u evaluates to one.

For correctness, it can be shown by induction that the output r_0 of the circuit G_n evaluates to one on input x if and only if M accepts starting from $C(r_0)$ and an empty stack [18].

Consider the space used by M on input $x \in L$ of length n. In checking a gate v, M must remember the gate v and its type. If v is an OR gate, M needs space to record information pertaining to a true input of v. This uses space $O(S(n))$. The space used for the gate v can be reused at the next level of recursion. If v is an AND gate, the information pertaining to all but one of its inputs is stored in the stack. This uses space $O(S(n))$. However, since the depth of the circuit is bounded by $O(\log T(n))$, the stack may have $O(\log T(n))$ such pieces of information, using altogether $O(S(n) \log T(n))$ space. The uniformity machine uses $O(S(n))$ space. Therefore, the total space used in the simulation by M is $O(S(n) \log T(n))$.

For the time bound of M, we first note that any accepting subtree of the circuit will have size $T^{O(1)}(n)$. The machine M, in verifying whether G_n accepts its input, traverses such an accepting tree in a depth-first fashion, visiting every vertex at most twice. For each node visited, M uses time $O(S(n))$ to guess the information pertaining to the node and time $O(S(n))$ to invoke the uniformity machine to verify its guesses. Recall that the uniformity machine is a deterministic machine using time $O(S(n))$. Since $S(n) \leqq T(n)$, the total time used by M is $T^{O(1)}(n)$. □

In the proof of Lemma 4 above, the space used for the stack can be completely avoided if the circuits being simulated are skew circuits. This observation leads immediately to the following lemma:

LEMMA 5. *For* $S(n) = \Omega(\log n)$, $T(n) = \Omega(n)$, *and* $S(n) \leqq T(n)$,

$$\text{Uniform Skew Circuit SIZE,DEPTH } (2^{O(S(n))}, T^{O(1)}(n)) \subseteq \text{NSPACE,TIME } (S(n), T^{O(1)}(n)).$$

Lemmas 2 and 5 yield the following theorem:

THEOREM 6. *For* $S(n) = \Omega(\log n)$, $T(n) = \Omega(n)$, *and* $S(n) \leqq T(n)$,

$$\text{NSPACE,TIME } (S(n), T^{O(1)}(n)) = \text{Uniform Skew Circuit SIZE,DEPTH } (2^{O(S(n))}, T^{O(1)}(n)).$$

The following characterizations of nondeterministic time using skew circuits and semi-unbounded fan-in circuits are now immediate from Lemmas 2, 3, and 4.

THEOREM 7. *For $T(n)=\Omega(n)$, the following complexity classes are equal*:
1. NTIME $(T^{O(1)}(n))$.
2. Uniform Skew Circuit DEPTH $(T^{O(1)}(n))$.
3. Uniform Semi-Unbounded Fan-in Circuit SIZE,DEPTH $(2^{O(T(n))}, \log T(n))$.

As interesting consequences of Theorems 6 and 7, we obtain the following Boolean circuit characterizations of the classes NLOG, PSPACE, and NP.

COROLLARY 8.
1. NLOG = Uniform Skew Circuit SIZE $(n^{O(1)})$.
2. PSPACE = Uniform Skew Circuit SIZE $(2^{n^{O(1)}})$.
3. NP = Uniform Skew Circuit DEPTH $(n^{O(1)})$.
4. NP = Uniform Semi-Unbounded Fan-in Circuit SIZE,DEPTH $(2^{n^{O(1)}}, \log n)$.

3. Other characterizations of nondeterministic time. This section contains some characterizations of nondeterministic time that follow as simple consequences of known results. We will consider only bounded fan-in Boolean circuits in this section. Perhaps the most interesting of the characterizations here is the one using the depth and degree measures for Boolean circuits. This suggests the characterization results in § 4 of counting classes on the basis of nondeterministic time bounded computations.

Ruzzo [13] showed that nondeterministic time $T(n)$ is the class of languages accepted by alternating Turing machines simultaneously using space $O(T(n))$ and tree size $O(T(n))$. Combined with the simulation by Ruzzo [13] of space and tree size bounded alternating Turing machines by space and time-bounded alternating Turing machines (used in the proof of Lemma 3), this provides a new characterization of nondeterministic time bounded classes on the alternating Turing machine model. The close relationship between Boolean circuits and alternating Turing machines [14] also leads to another Boolean circuit characterization of nondeterministic time in terms of size and tree size. Finally, the correspondence between degree and tree size for Boolean circuits (see Lemma 1) yields yet another Boolean circuit characterization of these classes in terms of degree and depth resources.

We will summarize these three characterizations in Theorem 9 below. The proof of this theorem can be reconstructed from the results mentioned.

THEOREM 9. *For $T(n)=\Omega(n)$, the following complexity classes are equal*:
1. NTIME $(T^{O(1)}(n))$.
2. Semi-Unbounded ATIME,ALTERNATIONS $(T^{O(1)}(n), \log T(n))$.
3. Uniform Circuit SIZE,TREESIZE $(2^{T^{O(1)}}(n), T^{O(1)}(n))$.
4. Uniform Circuit DEPTH,DEGREE $(T^{O(1)}(n), T^{O(1)}(n))$.

Thus, for instance, NP has the following characterization in terms of degree and depth of Boolean circuits:

COROLLARY 10. NP = Uniform Circuit DEPTH,DEGREE $(n^{O(1)}, n^{O(1)})$.

The Boolean circuit characterization of NP in Corollary 10 should be contrasted with the following bounded fan-in Boolean circuit characterization of PSPACE [2], [14]:

$$\begin{aligned}\text{PSPACE} &= \text{Uniform Circuit DEPTH } (n^{O(1)}) \\ &= \text{Uniform Circuit DEPTH,DEGREE } (n^{O(1)}, 2^{n^{O(1)}}).\end{aligned}$$

Constant depth circuits. Before concluding this section, we mention another definition of NP using constant depth unbounded fan-in circuits. We will show this

by exhibiting a uniform family of constant depth Boolean circuits for the conjunctive normal form satisfiability problem.

Let SAT denote the language consisting of all strings that are (reasonable) encodings of satisfiable conjunctive normal form formulas. Let all length r strings in SAT encode satisfiable formulas that have n variables and m clauses. The rth member G_r of a uniform circuit family $\{G_r\}$ that accepts SAT is described below (see Fig. 1).

• The output of G_r is an OR gate labeled $[0, n, m]$. This gate evaluates to one on input x if and only if the formula encoded by x is satisfiable.

• The OR gate $[0, n, m]$ has as inputs AND gates labeled $[1, n, m, j]$ for $0 \leqq j \leqq 2^n - 1$. An AND gate $[1, n, m, j]$ evaluates to one if and only if the input formula evaluates to one when the variables in the formula are assigned bit values from the integer j.

• Each AND gate labeled $[1, n, m, j]$ has as inputs OR gates labeled $[2, n, m, j, k]$ for $1 \leqq k \leqq m$. An OR gate $[2, n, m, j, k]$ evaluates to one if and only if the kth clause in the input formula evaluates to one when the variables in the formula are assigned bit values from the integer j.

• The inputs of an OR gate labeled $[2, n, m, j, k]$ are OR gates labeled $[3, n, m, j, k, p]$ for $1 \leqq p \leqq n$. An OR gate $[3, n, m, j, k, p]$ is the output of a subcircuit that evaluates to one if and only if the pth variable occurs in the kth clause as a positive (negative) literal and the pth bit of j is one (respectively, zero). If the pth variable does not occur in clause k, then a gate of the form $[3, n, m, j, k, p]$ evaluates to zero.

The family of Boolean circuits have size $O(m2^n)$ and constant depth. The OR gates have fan-in at most 2^n and the AND gates have fan-in at most m. It can be verified that the direct connection language for $\{G_r\}$ can be recognized by a deterministic

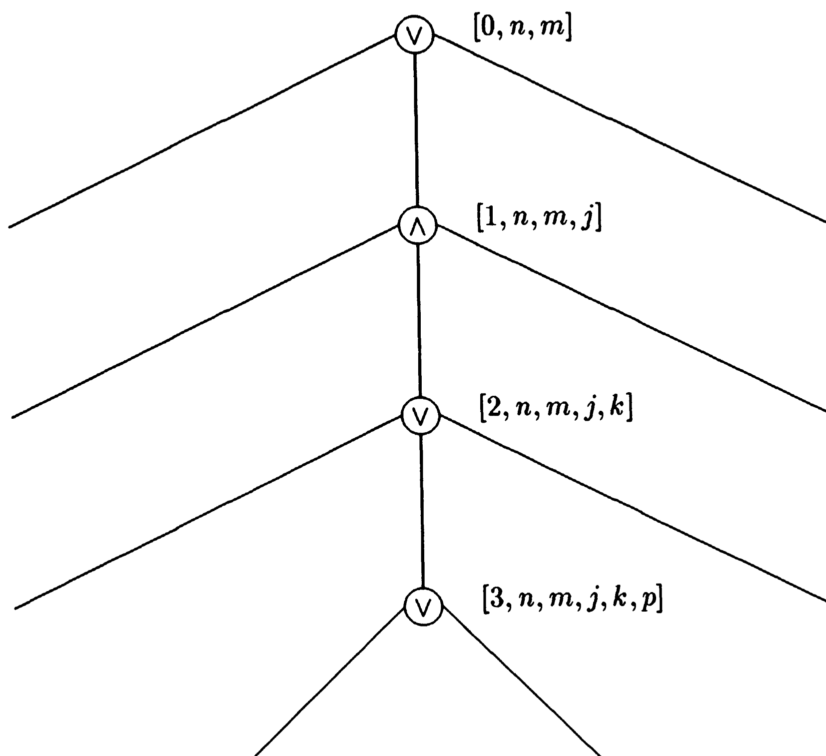


FIG. 1. *Constant depth unbounded fan-in circuits for CNF satisfiability.*

Turing machine in polynomial time, thus showing that this is a uniform family of circuits.

4. Monotone arithmetic circuits and counting classes. This section contains the characterizations of counting classes based on nondeterministic time bounded computations on the monotone arithmetic circuit model. A monotone arithmetic circuit is an arithmetic circuit that uses only the addition and multiplication operators and whose inputs are nonnegative integers. We will also characterize these classes in terms of the number of accepting subtrees in the Boolean circuit model. As corollaries, we obtain characterizations of the class #P on these models.

4.1. DEFINITIONS. It will be convenient to consider Boolean circuits in which every AND gate has exactly two inputs.

Monotone arithmetic circuits. These are defined just as Boolean circuits, except that the gates compute the sum and product of their inputs instead of computing the OR and AND functions. Although the results in this section, especially Lemma 13, can be strengthened to handle n-bit nonnegative integers as inputs to the circuit, it suffices to consider only single-bit inputs.

We will denote a gate computing the sum (product) of its inputs as a PLUS (respectively, MULT) gate.

Uniformity. We will slightly modify the definition of uniformity in § 1.2 to do a parsimonious simulation in Lemma 15.

Define the *direct connection language* of a family $\{G_n\}$ of Boolean circuits to be the set of strings of the form $\langle n, g, y, p\rangle$ such that either (i) g is an OR gate and y is an input of g or (ii) g is an AND gate and y is a left (right) input of g if p is L (respectively, R) or (iii) g is a gate name and y is the type of the gate g. A family $\{G_n\}$ of Boolean circuits of size $C(n)$ is said to be *uniform* if the corresponding direct connection language can be recognized by a deterministic Turing machine in time $O(\log C(n))$.

The uniformity condition for monotone arithmetic circuits is defined exactly as for Boolean circuits with PLUS (MULT) gates replaced for OR (respectively, AND) gates.

Degree. The degree measure for monotone arithmetic circuits is defined analogous to Boolean circuits (see § 1.2). Thus, the constants have degree zero, the circuit inputs have degree one, the degree of a PLUS vertex is the maximum of the degree of its inputs, and the degree of a MULT vertex is the sum of the degrees of its inputs.

Notations. Let $\mathcal{N}$ denote the set of natural numbers.

A function $f:\{0,1\}^* \rightarrow \mathcal{N}$ is in #Uniform Circuit SIZE,DEPTH, DEGREE $(Z(n), d(n), D(n))$ if and only if there exists a uniform family $\{G_n\}$ of Boolean circuits of size $O(Z(n))$, depth $O(d(n))$, and degree $O(D(n))$ such that for all strings x of length n, $f(x)$ is the number of accepting subtrees of G_n on input x.

The other counting classes are defined in a similar fashion.

4.2. THE CHARACTERIZATION RESULTS. The following fact can be used to set up a correspondence between Boolean and monotone arithmetic circuits. The proof of this fact is a direct consequence of the definition of an accepting subtree of a Boolean circuit (see § 1.2).

FACT 11. *Let B be a Boolean circuit that evaluates to one on input x. Given x as an input, the number of accepting subtrees of B rooted at an* OR (AND) *gate v is the sum (respectively, product) of the number of accepting subtrees of B rooted at the inputs of v.* □

It may be noted that Lemmas 12, 13, and 14 below are stronger statements than needed to prove the main results of this section, namely, Lemma 15 and Theorem 17.

LEMMA 12. *Let B be a Boolean circuit of size Z, depth d, and degree D. Then there exists an arithmetic circuit A of size Z, depth d, and degree D such that B has p accepting subtrees on an input x on which it evaluates to one if and only if A has value p on input x.*

Proof sketch. Given a Boolean circuit B, let the arithmetic circuit A be obtained by replacing all the OR (AND) gates of B by PLUS (respectively, MULT) gates. Then the conclusion follows by using Fact 11. □

LEMMA 13. *Let A be a monotone arithmetic circuit of size Z, depth d, and degree D with n inputs from $\{0, 1\}$. Then there exists a Boolean circuit B of size Z, depth d, and degree D such that A has value p if and only if B has p accepting subtrees given this input.*

Proof sketch. Given a monotone arithmetic circuit A, the Boolean circuit B is obtained from A by replacing all PLUS (MULT) gates by OR (respectively, AND) gates. The proof follows by a simple inductive argument. □

The circuits involved in Lemmas 12 and 13 can be made uniform, thereby showing the following correspondence between monotone arithmetic circuits and Boolean circuits.

LEMMA 14. *For $Z(n)$, $D(n) = \Omega(n)$,*

$$\#\text{Uniform Circuit SIZE,DEPTH,DEGREE } (Z^{O(1)}(n), d(n), D(n))$$
$$= \text{Uniform Monotone Arithmetic Circuit SIZE, DEPTH, DEGREE } (Z^{O(1)}(n), d(n), D(n)).$$

Lemma 15 below establishes the correspondence between the number of accepting paths in nondeterministic Turing machines and the number of accepting subtrees of Boolean circuits.

LEMMA 15. *For $T(n) = \Omega(n)$,*

$$\#\text{NTIME } (T^{O(1)}(n)) = \#\text{Uniform Circuit DEPTH,DEGREE } (T^{O(1)}(n), T^{O(1)}(n)).$$

Proof. Let M be a nondeterministic Turing machine that runs in time $T(n)$. By Theorem 7, there exists a uniform family $\{B_n\}$ of $O(T(n))$ depth bounded skew circuits that accepts the same language as M. The degree of B_n is $O(T(n))$. This is due to the fact that the degree of a depth d skew circuit cannot exceed d. Any accepting subtree of B_n, given an input on which it evaluates to one, is a completely skewed binary tree. We claim that M has p accepting paths on an input x of length n if and only if B_n has p accepting subtrees.

To simplify the proof, we will assume that M is deterministic while reading its inputs and that the immediate successor of a read configuration is an existential configuration.

Let x be an input of length n accepted by M. Then B_n evaluates to one on x. We will show that there is a bijective function that maps the accepting paths in the computation tree of M on input x with the accepting subtrees of B_n on input x.

Let p be an accepting path of M on input x. The starting vertex of p is labeled by the initial configuration c_I of M. Consider the following subtree $A(p)$ of B_n on input x. The root of $A(p)$ is the output gate $[0, c_I]$ of B_n. In general, the construction proceeds as follows. For the tth vertex of p labeled with an existential configuration c, pick the corresponding gate $[t, c]$ of B_n. The configuration d that immediately succeeds c along p is either an existential configuration or a read configuration. If d is an existential configuration, pick as the input of the gate $[t, c]$ its input labeled $[t+1, d]$. Suppose d is a read configuration in which M reads the ith input symbol and moves to an existential configuration $e(f)$ if the ith input is zero (respectively, one). Consider the case when the ith input symbol is zero. (The construction in the case when the ith input symbol is one is analogous.) Then d has the configuration e

as its immediate successor along p. Pick the gate $[t+1, d, i]$ as the input of the gate $[t, c]$, the AND gate $[t+1, d, i, 0]$ as the input of $[t+1, d, i]$, and the gate $[t+2, e]$ as the input of the gate $[t+1, d, i, 0]$. It is easy to see that $A(p)$ is an accepting subtree of B_n on input x.

The mapping described above from accepting paths of M on input x to accepting subtrees of B_n on input x is well defined. We will now argue that it is also a bijective function.

Suppose p and q are two distinct accepting paths of M on input x. Let $A(p)$ and $A(q)$ be the corresponding subtrees defined by the above mapping. Now, p and q both have the same start vertex, namely, the one labeled with the initial configuration c_I. Let the initial common segment of p and q have t vertices. Let the tth vertex be labeled by the configuration c. Then c must be an existential configuration. The corresponding gates in $A(p)$ and $A(q)$ are labeled by $[t, c]$. Since the immediate successor of c in p is different from that of c in q, the input of the gate $[t, c]$ in $A(p)$ is different from that of $[t, c]$ in $A(q)$.

Suppose A is an accepting subtree of B_n on input x of length n. We claim that there is an accepting path p of M on input x such that A is the image of p as defined by the mapping above. The path p is constructed as follows. The starting vertex of p is labeled with the initial configuration c_I. Let $[t, c]$ be a vertex in A where c corresponds to an existential configuration of M on input x. There are two cases.

Case 1. Suppose the gate $[t+1, d]$ is included in A as the input of the gate $[t, c]$. Then d is an existential configuration and it is an immediate successor of the configuration c of M. Since A is an accepting subtree on input x, the gate $[t+1, d]$ evaluates to one on input x. It follows that d is an accepting configuration of M on input x. Include a vertex labeled d as the immediate successor of the vertex labeled c along p.

Case 2. Suppose the gate $[t+1, d, i]$ is included in A as the input of the gate $[t, c]$. Then d is a read configuration that is an immediate successor of c. If $[t+1, d, i, 0]$ ($[t+1, d, i, 1]$) is the input of $[t+1, d, i]$ that is included in A, the ith input symbol must be zero (respectively, one). Consider the case when the ith input symbol is zero. (The case when the ith input symbol is one is analogous.) Let the input of $[t+1, d, i, 0]$ included in A be the gate $[t+2, e]$. Then include the vertex labeled d as the immediate successor of c and the vertex labeled e as the immediate successor of d along p. It is easy to verify that p is an accepting path of M on input x and A is an image of p defined by the above mapping.

Conversely, let $\{B_n\}$ be a uniform family of Boolean circuits of depth $T^{O(1)}(n)$ and degree $T^{O(1)}(n)$. Let M be a nondeterministic Turing machine that simulates B_n on an input x of length n in a depth-first fashion, as in the proof of Lemma 4. The one difference here is the need to ensure that the simulation of an AND gate maintains the correspondence between the number of accepting paths of the machine and the number of accepting subtrees of the circuit. Let $C(v)$ denote the configuration of M as it begins checking the gate v.

In simulating an AND gate v, M does the following. It guesses the right input, say v_2, of v, verifies with the uniformity machine that the guess is correct, and pushes v_2 onto the stack. It then guesses the left input, say v_1, of v, verifies with the uniformity machine that the guess is correct, and verifies that v_1 evaluates to one. This will guarantee that there is a single accepting path segment from the configuration $C(v)$ to the configuration $C(v_1)$.

Then it follows, from the claim below, that M has p accepting paths on x if and only if B_n has p accepting subtrees on input x.

CLAIM. *Let v be a vertex in B_n that evaluates to one on input x. If M begins its simulation at v, it has p accepting paths rooted at $C(v)$ if and only if there are p accepting subtrees of B_n rooted at v.*

Proof of the claim. This is by induction on the depth $d(v)$ of the vertex v.

The claim is clearly true for an input vertex v with value one.

Suppose v is an OR gate that evaluates to one on x. Let $v_1, \cdots, v_m$ be its inputs. Let $1 \leqq q \leqq m$ of these inputs, say $v_{i1}, v_{i2}, \cdots, v_{iq}$, evaluate to one on input x. The machine M, in checking whether v evaluates to one, existentially chooses one of these q inputs. Thus, the number of accepting paths rooted at $C(v)$ is given by the sum of the number of accepting paths rooted at $C(v_{i1}), C(v_{i2}), \cdots, C(v_{iq})$. By induction hypothesis, this sum is equal to the sum of the accepting subtrees rooted at $v_{i1}, v_{i2}, \cdots, v_{iq}$. Since this is equal to the number of accepting subtrees of B_n rooted at v, the claim follows. □

Suppose v is an AND gate that evaluates to one on x. Let v_1 and v_2 be its inputs. By construction, the number of accepting paths rooted at $C(v)$ is equal to the number of accepting paths rooted at $C(v_1)$. That is, if M begins its simulation with the gate v, there is a single accepting path segment from $C(v)$ to $C(v_1)$. Thus, the number of accepting paths rooted at $C(v)$ is the same as the number of accepting paths rooted at $C(v_1)$. The machine M, in verifying v_1, traverses an accepting subtree of B_n rooted at v_1. It then pops the vertex v_2. Hence, there is a vertex labeled $C(v_2)$ along every accepting path of M rooted at $C(v_1)$. Therefore, the number of accepting paths rooted at $C(v_1)$ is the product of the number of accepting path segments from $C(v_1)$ to $C(v_2)$ and the number of accepting paths rooted at $C(v_2)$. By induction hypothesis, the number of accepting path segments from $C(v_1)$ to $C(v_2)$ is the number of accepting subtrees rooted at v_1 of B_n, and the number of accepting paths rooted at $C(v_2)$ is the number of accepting subtrees of B_n rooted at v_2. It follows that the number of accepting paths rooted at $C(v)$ is the number of accepting subtrees rooted at v of B_n.

By Lemma 1, the tree size of B_n is $T^{O(1)}(n)$. Since B_n has size at most exponential in $T^{O(1)}(n)$, it follows, as in the simulation of Lemma 4, that M uses time $T^{O(1)}(n)$. □

In Lemma 15 we could have used semi-unbounded fan-in circuits instead of bounded fan-in circuits to obtain the following result:

THEOREM 16. *For $T(n)=\Omega(n)$,*

$$\#\text{NTIME}\,(T^{O(1)}(n)) = \#\text{Uniform Semi-Unbounded Fan-in Circuit SIZE,DEPTH,DEGREE}\,(2^{T^{O(1)}(n)}, T^{O(1)}(n), T^{O(1)}(n)).$$

Lemmas 14 and 15 together imply the following theorem:

THEOREM 17. *For $T(n)=\Omega(n)$,*

$$\#\text{NTIME}\,(T^{O(1)}(n)) = \text{Uniform Monotone Arithmetic Circuit DEPTH,DEGREE}\,(T^{O(1)}(n), T^{O(1)}(n)).$$

As a special case of the above theorem, we obtain the following new characterization of the important counting class $\#P$:

COROLLARY 18.

$$\#P = \text{Uniform Monotone Arithmetic Circuit DEPTH,DEGREE}\,(n^{O(1)}, n^{O(1)}).$$

4.3. SOME CONSEQUENCES. In this section, we will examine some consequences of the results in § 4.2.

Unique SAT. The unique SAT problem is defined as follows [10]: Given an instance of SAT, does it have a unique solution? As another interesting corollary of

Theorem 17, we can identify an arithmetic circuit value problem that is equivalent to the unique SAT problem.

Let M be a fixed uniformity machine for a family $\{G_n\}$ of monotone arithmetic circuits of polynomial depth and polynomial degree. Given as input n and an n-bit vector x, the MCVP1 problem is to determine whether the circuit G_n evaluates to one on input x.

COROLLARY 19. *There is a log space transformation from unique* SAT *to* MCVP1 *and vice versa.*

New NP*-complete problems.* Theorem 17 suggests a new arithmetic circuit value problem that is complete for NP. Let M be a fixed uniformity machine for a family $\{G_n\}$ of monotone arithmetic circuits of polynomial depth and polynomial degree. Given as input n and an n-bit vector x, the MCVP problem is to determine whether the circuit G_n evaluates to a nonzero value on input x.

PROPOSITION 20. *The MCVP problem is* NP *complete.*

Characterizing #PSPACE *by using monotone arithmetic circuits.* By using the known characterization of Boolean circuit depth by alternating time [14], the following analogue of Lemma 15 can be proven with the techniques in the proof of that lemma:

LEMMA 21. *For* $T(n) = \Omega(\log n)$,

$$\#\text{ATIME}\,(T^{O(1)}(n)) = \#\text{Uniform Circuit DEPTH}\,(T^{O(1)}(n)).$$

Combined with Lemma 14 and the result by Ladner [9] that #PSPACE = #ATIME $(n^{O(1)})$, Lemma 21 implies the following theorem:

THEOREM 22.

$$\#\text{PSPACE} = \text{Uniform Monotone Arithmetic Circuit DEPTH}\,(n^{O(1)}).$$

It should be noted here that Bertoni et al. [1] also characterized #PSPACE as the class of functions computed by polynomial time random access machines with the operations of addition, integer subtraction, multiplication, and integer division.

5. Conclusion. This work provides a circuit framework in which some well-known open problems of complexity theory can be studied. We considered two constraints on the Boolean circuit model, namely, skewness and semi-unboundedness, and we used this model to define nondeterministic space and time complexity classes. We also considered monotone arithmetic circuits to define counting classes based on nondeterministic time.

The known uniform Boolean circuit characterizations of classes between LOGCFL and PSPACE are summarized in Table 1 (the definitions of the classes LOGCFL and

TABLE 1
Circuit definitions of complexity classes.

Class	OR fan-in	AND fan-in	Size	Depth	Degree
LOGCFL	$n^{O(1)}$/bounded	$n^{O(1)}$/bounded	$n^{O(1)}$		$n^{O(1)}$
	$n^{O(1)}$	bounded	$n^{O(1)}$	$\log n$	
AC^1	$n^{O(1)}$	$n^{O(1)}$	$n^{O(1)}$	$\log n$	
P	$n^{O(1)}$/bounded	$n^{O(1)}$/bounded	$n^{O(1)}$		
NP	$2^{n^{O(1)}}$	bounded	$2^{n^{O(1)}}$	$\log n$	
	$2^{n^{O(1)}}$/bounded	$2^{n^{O(1)}}$/bounded	$2^{n^{O(1)}}$	$n^{O(1)}$	$n^{O(1)}$
PSPACE	$2^{n^{O(1)}}$/bounded	$2^{n^{O(1)}}$/bounded	$2^{n^{O(1)}}$		$2^{n^{O(1)}}$
	$2^{n^{O(1)}}$/bounded	$2^{n^{O(1)}}$/bounded	$2^{n^{O(1)}}$	$n^{O(1)}$	

P in this table use log space uniformity). It should not be too difficult to construct entries for classes above PSPACE.

As a consequence of these characterizations, we can define for each of these complexity classes a Boolean circuit value problem that is a natural complete problem for the class. For example, the following circuit value problem is NP-complete. Let M be a fixed uniformity machine for a family $\{G_n\}$ of Boolean circuits of polynomial depth and polynomial degree. Given as input n and an n-bit vector x, the problem is to determine whether the circuit G_n evaluates to one on input x.

We will conclude with a few remarks about the relevance of the semi-unboundedness notion for questions in complexity theory. From Table 1, it can be seen that many of the well-known space and time complexity classes have definitions in terms of semi-unbounded fan-in circuits. For instance, the following are definitions of some well-known classes using the semi-unbounded fan-in circuit model:

$$\text{LOGCFL} = \text{Uniform Semi-Unbounded Fan-in Circuit SIZE,DEPTH } (n^{O(1)}, \log n);$$

$$P = \text{Uniform Semi-Unbounded Fan-in Circuit SIZE,DEPTH } (n^{O(1)}, n^{O(1)});$$

$$\text{NP} = \text{Uniform Semi-Unbounded Fan-in Circuit SIZE,DEPTH } (2^{n^{O(1)}}, \log n);$$

$$\text{PSPACE} = \text{Uniform Semi-Unbounded Fan-in Circuit SIZE,DEPTH } (2^{n^{O(1)}}, n^{O(1)}).$$

One can define an analogue of the polynomial time hierarchy by using semi-unbounded alternating Turing machines. Then, by Theorem 9, NP is the class of languages accepted by polynomial time semi-unbounded alternating Turing machines using $O(\log n)$ alterations. This is interesting because it shows that with the constraint of semi-unboundedness $O(\log n)$ alternations are in NP, whereas without this constraint, even constant alternations are not known to be in NP.

Acknowledgments. I am grateful to Martin Tompa for useful discussions. My thanks also to Larry Ruzzo, whose work on alternating Turing machines and Boolean circuits was a source of inspiration for the results reported here. I am also thankful to Gary Peterson for his comments.

REFERENCES

[1] A. BERTONI, G. MAURI, AND N. SABADINI, *Simulations among classes of random access machines and equivalence among numbers succinctly represented*, Ann. Discrete Math., 25 (1985), pp. 65–90.

[2] A. BORODIN, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–743.

[3] A. BORODIN, S. A. COOK, P. W. DYMOND, W. L. RUZZO, AND M. TOMPA, *Two applications of inductive counting for complementation problems*, SIAM J. Comput., 18 (1989), pp. 559–578.

[4] S. A. COOK, *Deterministic CFL's are accepted simultaneously in polynomial time and log squared space*, in Proc. 11th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1979, pp. 338–345.

[5] ———, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2–22.

[6] P. W. DYMOND AND S. A. COOK, *Complexity theory of parallel time and hardware*, Inform. and Comput., 80 (1989), pp. 205–226.

[7] L. M. GOLDSCHLAGER, *The monotone and planar circuit value problems are log space complete for P*, SIGACT News, 9 (1977), pp. 25–29.

[8] R. E. LADNER, *The circuit value problem is log space complete for P*, SIGACT News, 7 (1975), pp. 18–20.

[9] ———, *Polynomial space counting problems*, SIAM J. Comput., 18 (1989), pp. 1087–1097.

[10] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *The complexity of facets (and some facets of complexity)*, J. Comput. System Sci., 28 (1984), pp. 244–259.

[11] N. PIPPENGER, *On simultaneous resource bounds*, in Proc. 20th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1979, pp. 307–311.

[12] N. PIPPENGER AND M. J. FISCHER, *Relations among complexity measures*, J. Assoc. Comput. Mach., 26 (1979), 361–381.

[13] W. L. RUZZO, *Tree-size bounded alternation*, J. Comput. System Sci., 20 (1980), pp. 218–235.

[14] ———, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

[15] S. SKYUM AND L. G. VALIANT, *A complexity theory based on Boolean algebra*, J. Assoc. Comput. Mach., 32 (1985), pp. 484–502.

[16] L. STOCKMEYER AND U. VISHKIN, *Simulation of parallel random access machines by circuits*, SIAM J. Comput., 13 (1984), pp. 409–422.

[17] L. G. VALIANT, *The complexity of computing the permanent*, Theoret. Comput. Sci., 8 (1979), pp. 189–201.

[18] H. VENKATESWARAN, *Properties that characterize LOGCFL*, J. Comput. System Sci., 42 (1991), pp. 380–404.

[19] H. VENKATESWARAN AND M. TOMPA, *A new pebble game that characterizes parallel complexity classes*, SIAM J. Comput., 18 (1989), pp. 533–549.

SIAM J. COMPUT.
Vol. 21, No. 4, pp. 671-696, August 1992

© 1992 Society for Industrial and Applied Mathematics
004

AN OPTIMAL ALGORITHM FOR INTERSECTING THREE-DIMENSIONAL CONVEX POLYHEDRA*

BERNARD CHAZELLE†

Abstract. This paper describes a linear-time algorithm for computing the intersection of two convex polyhedra in 3-space. Applications of this result to computing intersections, convex hulls, and Voronoi diagrams are also given.

Key words. computational geometry, convex polyhedra

AMS(MOS) subject classifications. 68Q25, 68H05

1. Introduction. Given two convex polyhedra in 3-space, how fast can we compute their intersection? Over a decade ago, Muller and Preparata [22] gave the first efficient solution to this problem by reducing it to a combination of intersection detection and convex hull computation. Another route was followed in 1984 by Hertel et al., who solved the problem by using space sweep [16]. In both cases, a running time of $\Theta(n \log n)$ was achieved, where n is the combined number of vertices in the two polyhedra. Resolving the true complexity of the problem, however, remained elusive.

The different but related problem of *detecting* whether two convex polyhedra intersect, by using preprocessing, was studied by Chazelle and Dobkin [5], Dobkin and Munro [9], and Dobkin and Kirkpatrick [6]. More germane to our concerns here is the off-line version of the detection problem. Dobkin and Kirkpatrick [7] have shown that detecting whether two convex polyhedra intersect can be done in a linear number of operations. By stating the problem as a linear program over three variables, other linear-time algorithms originate in the works of Megiddo [19] and Dyer [10]. Previous results also include an efficient algorithm for intersecting two polyhedra, one of which is convex (Mehlhorn and Simon [21]). Optimal solutions for intersecting convex polygons are given in Shamos and Hoey [27] and O'Rourke et al. [23]. For additional background material on polyhedral intersections, the reader should consult Edelsbrunner [11], Mehlhorn [20], and Preparata and Shamos [25].

Our main result is an algorithm for constructing the intersection between two convex polyhedra in linear time. The algorithm does not use any complicated data structure and seems a good candidate for practical implementation. As is customary, our result assumes that the input conforms with any one of the standard (linear-time equivalent) polyhedral representations given in the literature [3], [15], [22]. This is not a minor point, because nonstandard representations can easily make the problem more difficult. (For example, think how much more difficult the problem would be if we were given only the vertices without any other facial information.) From our algorithm for pairwise intersections we immediately derive an efficient method for intersecting k convex polyhedra. The complexity of the algorithm is $O(n \log k)$, where n is the total number of vertices, which is provably optimal. Other applications include merging Voronoi diagrams in two dimensions and computing convex hulls in 3-space.

2. Polyhedra and shields. At the heart of Dobkin and Kirkpatrick's detection [6] and separation algorithms [7] is an ingenious hierarchical representation of a convex

* Received by the editors February 15, 1989; accepted for publication (in revised form) July 16, 1991. The author wishes to acknowledge the National Science Foundation for supporting this research in part under grant CCR-8700917.

† Department of Computer Science, Princeton University, Princeton, New Jersey 08544.

polyhedron. Further applications of that versatile data structure have been given in [12], [21]. The representation can be seen as a specialization of Kirkpatrick's point location structure [18]. A convex polyhedron P of n vertices is made the first element of a descending chain of $O(\log n)$ nested convex polyhedra, such that the last one has a constant number of vertices and the others differ from their immediate predecessors by shelling off small, disjoint polyhedral cones. We need to go further and modify this hierarchy of polyhedra in several ways.

First, we represent the set of nested polyhedra as a single geometric object, namely, a simplicial cell complex, so we can walk freely from one to the next. In this context, walking means being able to trace a polygonal curve in 3-space within the hierarchy in time proportional to the size of the curve (i.e., its number of vertices) and the number of cells (counting multiplicities) crossed by the curve. Thus, if a curve lies inside P and connects two points on the boundary, we can go from one endpoint to the other while keeping track of where we are within the hierarchy, all of this in time proportional to the size of the curve and the number of cells crossed. If the curve is a straight-line segment, then because of convexity the time becomes $O(\log n)$.

This data structure is still insufficient for our purposes, because sometimes we will need to follow a curve that leaves P and later re-enters the polyhedron from the outside. To trace the curve after we leave P we need a hierarchy for the "outside" of P as well. Unfortunately, the outside of a convex polyhedron is not convex, so we cannot apply the Dobkin–Kirkpatrick construction verbatim. Instead, we switch to dual space because the set of planes that avoid P gets mapped to a convex polyhedron. So, we now have two nested sequences of $O(\log n)$ polyhedra, one fitting inside P and the other fitting inside its dual. The resulting data structure is called the *shield* of P: It consists of a primal part, which allows us to navigate inside P, and a dual part, which, we hope, allows us excursions outside. The latter is true, but in an indirect way. Indeed, to trace a curve that connects two points on the boundary of P and lies outside the polyhedron is still not very easy. But, instead, consider a finite sequence of planes, all of which lie outside P, except for the first and last ones, which are tangent to P. We can visualize this sequence mechanically by starting with the first plane and pivoting along the appropriate line to get to the second plane, and so on, until we reach the position of the last plane. Dually, this motion corresponds to the traversal of a polygonal curve inside the dual of P that connects two points on the boundary (namely, the duals of the first and last planes in the sequence). Because of the dual hierarchy we are thus able to trace this curve, which, in primal space, means "tracing" the corresponding sequence of planes. These operations will allow us to navigate inside and outside P and discover the points where we leave and re-enter the polyhedron. The navigation inside P follows polygonal curves, while the one outside P follows sequences of pivoting planes.

As it turns out, we cannot afford to keep the full contents of the hierachies: Only their outermost layers can be used. Thus, we discard all but a constant number, say k, of the nested polyhedra. The result is a geometric structure that we call the *k-shield* of P. Intuitively, we cannot afford to traverse either hierarchy all the way across because we would pay a factor of $\log n$ time in doing so, and this overhead would result in an $O(n \log n)$-time intersection algorithm. Of course, we are now missing so much information that navigating in a k-shield is rather difficult. However, we can use a well-tuned form of recursion to get around this problem.

Informally, the linear algorithm works as follows: We check that both input polyhedra P and Q have a point in common and we compute their k-shields, for some appropriate constant k. Then we begin to traverse the edges of one of the polyhedra,

say P, and while doing so we keep track of where we are in the primal part of the k-shield of Q (assuming that we start somewhere inside Q). This is called *broadcasting* from P. As long as we navigate inside Q we can use the primal part of its k-shield to guide us. When we reach portions of the boundary of P that lie outside Q, however, we must switch to dual space and use the dual part of the shield to guide the navigation. A transition from primal to dual space is called a *mutation*: It involves changing the mode of navigation from one that traces a polygonal curve within a shield to one that follows a sequence of pivoting planes or, equivalently, one that traces a polygonal curve in the dual part of the shield. Unfortunately, a mutation cannot be carried out instantaneously and requires a little bit of geometric work.

A yet more serious difficulty is what to do when we reach the last layer L in either one of the hierarchies of Q, say the primal one, and we need to go deeper to carry on the navigation. Recall that most of the inner layers of the shield have been removed and, thus, tracing a polygonal curve all across the hierarchy is not possible. When this happens we call upon the intersection algorithm recursively with P and L as input and thus discover, in this indirect manner, the tracing pattern along L. In other words, we use recursion to palliate the lack of inner layers. What makes this idea work is that as we do so we also switch from a broadcasting from P to a broadcasting from L. This switching trick is actually the key to breaking the $n \log n$ barrier. Indeed, this leads to a recurrence on the running time $T(n)$ of the form $T(n)=4T(n/5)+O(n)$, which solves to linear.

An interesting side effect is that because we operate in both primal and dual spaces, the algorithm ends up computing the intersection, as well as the convex hull, of the two input polyhedra. Actually, we keep switching between these two tasks in a co-routine-like fashion. Although the algorithm is not particularly complicated, proving its correctness requires a certain amount of thoroughness in investigating the topology of several convex polyhedra. The fact that polyhedral boundaries are not smooth manifolds further complicates the analysis but also makes it more interesting.

A. BACKGROUND. We begin with some geometric terminology. Given $X \subseteq \mathbb{R}^d$, the closure (respectively, interior) of X is denoted cl X (respectively, int X). The frontier of X is defined as $\mathrm{cl}\, X \cap \mathrm{cl}\,(\mathbb{R}^d \backslash X)$, or as $\mathrm{cl}\, X \cap \mathrm{cl}\,(A \backslash X)$ if the relative topology of some $A \supseteq X$ is understood. The unit d-sphere and the open unit d-ball are denoted S^d and D^d, respectively. A disjoint union of k-faces (subsets of $\mathbb{R}^d$ homeomorphic to D^k), for $k=0, \cdots, d$, is called a d-dimensional *cell complex* if, given any two faces f and g, the intersection of cl f and cl g is either a union of faces or the empty set. A cell complex is called *simplicial* (or triangulation) if each face is the interior of a simplex (in the relative topology of its affine closure).

We take a rather general view of a polyhedron as any subset of 3-space that is locally a cone with a compact base [26]. Given a point $p \in \mathbb{R}^3$ and a subset $C \subset \mathbb{R}^3$, we say that the points $\alpha p+(1-\alpha)q$, for all $q \in C$ and $0 \leqq \alpha \leqq 1$, form a *cone pC* if for each point of pC distinct from p, the choice of q is unique. A subset P of $\mathbb{R}^3$ is called a *polyhedron* if each point $p \in P$ has a cone neighborhood pC, whose *base* C is compact. We use the term *boundary* to refer to the frontier of a polyhedron P and denote it ∂P. It is not hard to see that a polyhedron is a locally finite union of simplices and, hence, is piecewise linear. It need not be a manifold, however. An example of a valid polyhedron is shown in Fig. 2.1. An open halfplane π is a polyhedron but, because of the compactness condition, ceases to be one as soon as we include one point on its frontier. Adding the entire frontier is fine, however.

We need this level of generality because we will sometimes be dealing with rather convoluted shapes. As it turns out, however, most of our time will be spent with convex

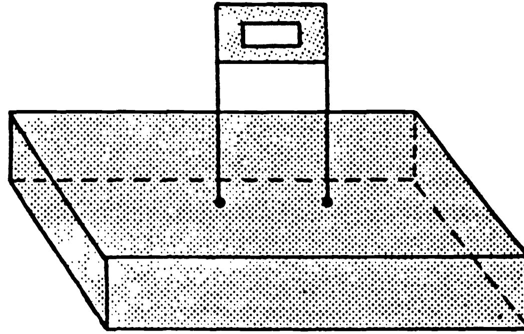


FIG. 2.1. *Example of a valid polyhedron.*

polyhedra, for which a more global (but slightly restrictive) definition is preferable [11]. A *convex polyhedron* is a nonempty intersection of a finite number of closed halfspaces. It is called a *convex polytope* if it is bounded. For technical convenience, we will restrict our discussion to convex polytopes, but it is easy to generalize it to the unbounded case. We assume that the boundary of a convex polytope P is facially structured as a two-dimensional cell complex with a minimal set of vertices. This last requirement means that a vertex (0-face) must be the intersection of three or more bounding planes (i.e., planes that delimit the defining halfspaces), but an edge (1-face) need not lie in the intersection of two distinct bounding planes. This assumption allows us to triangulate ∂P and still have a convex polytope; on the other hand, it forbids the facets (2-faces) incident upon any given vertex from being all coplanar. For storing two- and three-dimensional cell complexes we shall assume the representations of Baumgart [3], Muller and Preparata [22], or Guibas and Stolfi [15] and of Dobkin and Laszlo [8], respectively, or any other data structures that allow us to navigate at ease between adjacent cells. Such representations will be called *standard.*

To conclude this laundry list of assumptions and definitions, we introduce a well-known duality between points and planes, namely, the polarity δ, which maps any point $p=(\alpha, \beta, \gamma)$ distinct from the origin O to the plane of equation $\alpha x+\beta y+\gamma z=1$: $\delta(p)$ is the plane normal to Op that lies at a distance $1/|Op|$ from the origin on the same side as p. Given a convex polytope P, whose interior contains the origin, the dual of P is the set of planes whose dual points lie in P. Forming the union of all these planes and taking the closure of the complement defines a convex polytope, which is called the *dual polytope* of P and is denoted P^{δ}. If the polytope P has no coplanar facets (e.g., no triangulation has been forced upon its boundary), then each vertex, edge, and facet of P corresponds respectively to a unique facet, edge, and vertex of its dual polytope, and the correspondence is involutory. Given a standard representation of a convex polytope P, it is elementary to compute a standard representation of P^{δ} in linear time, supplemented with pointers between each k-face of P and its dual $(2-k)$-face. Note that if the origin is not interior to P, then instead of a single polytope we obtain one or two unbounded polyhedra in 3-space.

Central to the Muller–Preparata method [22] is the use of the fact that convex hulls and intersections play dual roles. Indeed, if the origin lies in the interior of two convex polytopes P and Q, then the convex hull of P^{δ} and Q^{δ} is the dual polytope of $P \cap Q$. In other words, identifying the binary operation *intersection* in primal space with the binary operation *convex hull* in dual space yields the following commutative

diagram:

$$\begin{array}{ccc} P, Q & \xrightarrow{\quad\delta\quad} & P^\delta, Q^\delta \\ \downarrow & & \downarrow \\ P \cap Q & \xrightarrow{\delta} & (P \cap Q)^\delta = \text{Hull}\,(P^\delta \cup Q^\delta) \end{array}$$

Unlike the Muller–Preparata approach, which commutes through the diagram only once, our algorithm will spend most of its time traveling back and forth between primal and dual spaces.

B. SHIELDING A CONVEX POLYTOPE. We open this discussion with a variant of the Dobkin–Kirkpatrick construction. Let P be a convex polytope of n vertices with nonempty interior. We assume that its boundary has been triangulated, which is easy to ensure in linear time. Recall that the *degree* of a vertex refers to its number of incident edges. We select a maximal independent set of constant-degree vertices: (i) Pick any vertex of degree at most 8, and mark it along with all its adjacent vertices; (ii) iterate on this process, always making sure to pick unmarked vertices. Termination occurs when we run out of unmarked vertices of degree at most 8. Because the number of edges is at most $3n-6$, we find that the sum of all vertex degrees does not exceed $6n-12$. Since every vertex has degree at least 3, the number m of vertices of degree at most 8 satisfies $9(n-m) \leqq 6n-12-3m$, and hence $m \geqq n/2$. Therefore, at least $n/18$ vertices will be selected in this process. As shown in Edelsbrunner [11], we can actually do better by considering vertices in order of nondecreasing degree. This allows us to find an independent set of at least $n/7$ vertices of degree at most 12. In both cases, the time for selecting the desired vertices is linear.

Around each selected vertex v, we perform some local surgery by removing v and its "umbrella" of incident faces and recomputing the convex hull of P (Fig. 2.2). Since v has degree at most 12, this shelling operation can be done in constant time. Note that because of the independence of the selected set of vertices, the order in which vertices are "popped out" does not matter. In $O(n)$ time we thus will have (i) removed all selected vertices and their incident faces, (ii) computed the new convex hull P_1, and (iii) triangulated its boundary. We easily verify that P_1 is a valid convex polytope; in particular, each of its vertices lies on at least three distinct bounding planes. We

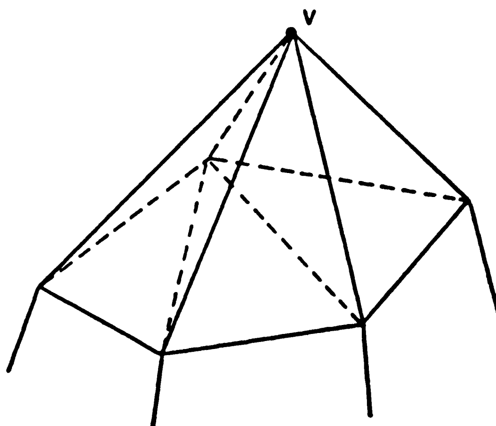


FIG. 2.2. *Removing v and its "umbrella" of incident faces.*

can now repeat this process with respect to P_1 and define a sequence of nested convex polytopes $P_0 \supset P_1 \supset \cdots \supset P_\alpha$, where (i) $P_0 = P$, (ii) P_α has constant size, and (iii) each cl $(P_i \backslash P_{i+1})$ is a collection of three-dimensional cones whose interiors are disjoint. If the interior of P_1 is empty, then at most two vertices were popped out and P has at most $12+2$ vertices. To avoid any difficulty, we do not bother with P_1 and set $\alpha = 0$ whenever P has at most 14 vertices. We use the same criterion to terminate the iteration.

Our next step is to compute a triangulation of P that is compatible with all the nested polytopes. This might be awkward to do during the shelling phase, because we may inherit the "wrong" triangulation from outer polytopes and create tetrahedra with empty interiors (Fig. 2.3). The difficulty is that a facet incident upon a popped-out vertex v of P_i might still contribute a portion of a facet of P_{i+1}. A simple fix is to retriangulate inside out. Assume that P_i has been given a triangulation compatible with $P_{i+1}, \cdots, P_\alpha$. Each cone of cl $(P_{i-1} \backslash P_i)$ can be triangulated directly by connecting its apex to the triangulation of its base provided by P_i. This will lead to a compatible triangulation of P in $O(n)$ time. Note that the resulting triangulation of ∂P might be different from the one we started with.

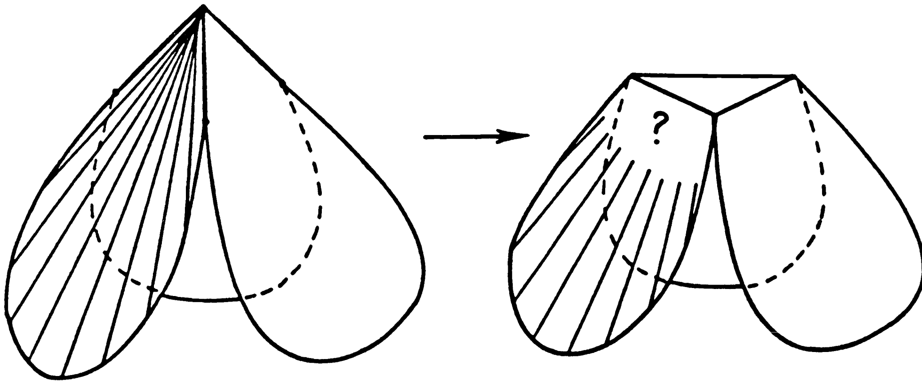


FIG. 2.3. *Inheriting the wrong triangulation.*

There is nothing startlingly novel here. The only slight twist from the Dobkin-Kirkpatrick construction is to transform the nested sequence into a three-dimensional triangulation. This will allow us to discover P from various angles by traversing it along straight lines, shooting rays through it, and in general exploring its geometry from within. Unfortunately, we also need to approximate P from the outside. What is unfortunate about this is that the complement of P is not convex, so we cannot play the same game over. Its dual polytope is convex, however, so it might just be the right time to jump into dual space.

After ensuring that no two facets are coplanar, by removing edges if necessary, we choose an origin O in the interior of P and we form its dual polytope P^δ. Then we triangulate ∂P^δ and submit P^δ to the in-growing process described above. This results in a sequence of nested convex polytopes $P^\delta = \Pi_0 \supset \Pi_1 \supset \cdots \supset \Pi_\beta$, where Π_β has constant size. The triangulation of $P_0 \backslash \text{int}\, P_\alpha$ is the *primal shield* of P; its counterpart between Π_0 and Π_β is called the *dual shield.* Unless specified otherwise, the term "shield" refers to both its primal and dual parts. Given any integer k such that $0 \leqq k \leqq \min\{\alpha, \beta\}$, the triangulations of $P_0 \backslash \text{int}\, P_k$ and $\Pi_0 \backslash \text{int}\, \Pi_k$ form the *k-shield* of P. Primal and dual k-shields are defined in the obvious way. All logarithms below are to the base 2.

LEMMA 2.1. *Let P be a convex polytope with nonempty interior, and let m be its total number of vertices and bounding planes. The shield of P can be constructed in $O(m)$ time. The total number of vertices and bounding planes in each P_k (and Π_k) is at most $3(1-1/7)^k m$. The total number of nested polytopes is less than $9 \log m + O(1)$.*

Proof. Let v_k and f_k be the number of vertices and bounding planes in P_k, respectively. The reason for dealing with $m_k = v_k + f_k$ is that this quantity is invariant under duality. Since the boundary of a convex polytope has Euler characteristic 2 and every facet has at least three incident edges, we derive $f_k \leqq 2v_k - 4$. Each pass in the algorithm removes at least one-seventh of the vertices; therefore, (i) the preprocessing is linear and (ii) $m_k \leqq 3(1-1/7)^k v_0 - 4$. □

C. NAVIGATING THROUGH A SHIELD. The usefulness of a shield owes to its being both an approximation scheme and a cell complex. Indeed, it gives us a "two-way" sequence of approximations for P, through which we can easily navigate and "discover" the boundary of P from any desired angle. This assumes that we use a proper representation, such as the Dobkin–Laszlo structure [8]. Without getting into the details of the data structure, let us just say that from each tetrahedron of a shield we can gain access to any of its four incident facets in constant time. Conversely, any facet leads us directly to its two incident tetrahedra. Also, the tetrahedra and facets incident upon an edge are accessible in cyclical order around the edge.

Let us consider a simple operation, such as being given a ray $\vec{r}$ with a starting point in a tetrahedron of, say, the primal shield of P, and being asked to traverse the primal shield along $\vec{r}$. In general, the ray will cut through a sequence of cells alternating between tetrahedra and triangles. When this is the case, there is no difficulty in discovering the sequences of cuts on the fly, at a cost of $O(1)$ per cut. Let us call a facet of the primal shield *primitive* if it lies on the boundary of one of the nested polytopes. Since the popped-out cones are bounded by primitive triangles, the ray $\vec{r}$ cannot cut more than a constant number of nonprimitive triangles in a row. Consequently, the total size of the cutting sequence is proportional to the number of primitive triangles intersected by the ray. A minor difficulty arises when the ray cuts through an edge or a vertex of the shield. In the case of an edge we are faced with two candidate triangles to be visited next. We can break ties by making arbitrary navigational conventions. For example, we might agree always to choose the triangle that is (locally) highest (or leftmost if there are several highest ones). We can also submit the ray $\vec{r}$ to a *symbolic* perturbation—see Edelsbrunner and Mücke [13] and Yap [28]. Note that we can easily generalize the mode of traversal to polygonal lines embedded in 3-space. The following summarizes our discussion.

LEMMA 2.2. *The complexity of traversing the primal (respectively, dual) shield along a polygonal line in three dimensions, knowing the starting cell, is proportional to the complexity of the polygonal line plus the number of nested polytopes P_i (respectively, Π_i) whose boundaries are cut during the traversal (counting multiplicities).*

3. Intersecting two convex polytopes. We begin with a brief discussion of what makes the problem not so easy. We can assume that we have a point O inside both convex polytopes P and Q, since such a witness (if any) can be discovered in linear time [7], [10], [19]. What remains to be done is in some sense merging P and Q. Imagine a sphere centered at O, on which ∂P and ∂Q are centrally projected. This gives us two spherical subdivisions S_P and S_Q. Merging the two subdivisions would do the job, but this might cause a quadratic blowup. The next "smartest" move might appear to be locating each vertex of S_P in S_Q and vice versa, which we can do in $O(n \log n)$ time, where n is the total number of vertices in P and Q [18]. Unfortunately,

it is easy to prove that this complexity is optimal. Clearly, we are still seeking too much information, and the subdivisions S_P and S_Q appear essentially worthless. So, let us bring shields into the picture. How about locating each vertex of P (respectively, Q) in the primal shield of Q (respectively, P)? Such information might be a good start from which to launch our intersecting attack. But actually this is still asking too much: Indeed, it can be proven that locating the vertices of P in the primal shield of Q requires $\Omega(n \log\log n)$ time in the worst case. All these attempts fail because we are working too far from the boundaries of P and Q and thus are giving free rein to our adversary. Pairwise intersections of convex polytopes possess a rich geometric structure into which we have yet to tap seriously. The time has come for a closer look at the geometry of the intersection problem.

A. BROADCASTING. We devote this section to defining the notion of broadcasting and showing that it is the essential operation in computing the intersection of two polytopes (Lemma 3.1). Let us restate our assumptions: P and Q are two convex polytopes with a total of n vertices, and their interiors contain the origin O. To simplify our discussion, we shall assume that P and Q have no two coplanar facets and are in general position relative to each other: No facet (respectively, edge) of one is coplanar (respectively, colinear) with a facet (respectively, edge) of the other, no vertex of one lies on the boundary of the other, etc. To borrow a cliché, relaxing these assumptions is tedious but not difficult.

Assume that the boundaries of P and Q have been triangulated by inheritance from their primal shields. Since the two polytopes are convex and contain the origin in their interiors, the boundary $\Sigma = \partial(P \cup Q)$ is the graph of $\max\{f, g\}$, where f and g are continuous functions $S^2 \mapsto \mathbb{R}^+$. It follows that Σ is homeomorphic to S^2. Let us now color ∂P blue and ∂Q red. (We apologize to the reader for not using a more evocative terminology: If it is any help for future reference, P and the first letter of "blue" sound somewhat alike.) Points that are both blue and red are said to be purple. The facets of Σ become monochromatic polygons. Beware: Some of them may be of nonconstant size, nonconvex, and even perforated. However, Σ can still be regarded as a two-dimensional cell complex. Of particular interest to us are the connected components of $\partial P \cap \partial Q$. These are disjoint, purple, simple cycles in the facial graph of Σ, which we call *laces*. removing all the laces from Σ creates relatively open polyhedral surfaces, called *regions* (white and dotted areas in Fig. 3.1).

Regions and laces partition Σ into maximal monochromatic connected subsets. Assume for the time being that Σ has at least one lace. Then the closure of a region

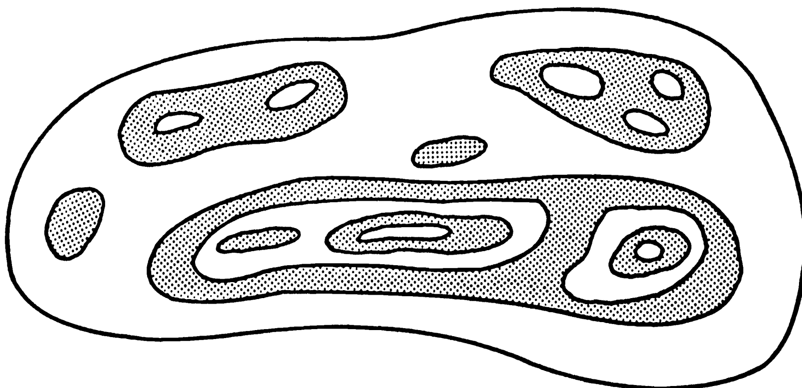


FIG. 3.1. *Creation of regions by removing all the laces from* Σ.

R is a (topological) manifold with boundary, which is homeomorphic to S^2 perforated by $k \geqq 1$ disjoint copies of D^2. Unlike some of the sets we will encounter later, this is a rather friendly one: It is an orientable bounded surface, its number of boundary components is k, and its Euler characteristic is $2-k$. The boundary of the manifold is also the frontier of R in the relative topology of Σ: By extension, we call it the *boundary* of R. The k connected components of the boundary are called the *bounding* laces of the region. Note that each lace of Σ bounds exactly two regions (of opposite color). A region R, being a monochromatic component of the graph max $\{f, g\}$, is paired with a homeomorphic component R^c of the graph min $\{f, g\}$: This component has the opposite color of the region R and is called its *co-region* (dotted area in Fig. 3.2—boundaries are left untriangulated for clarity). Here are more formal definitions of all these concepts.

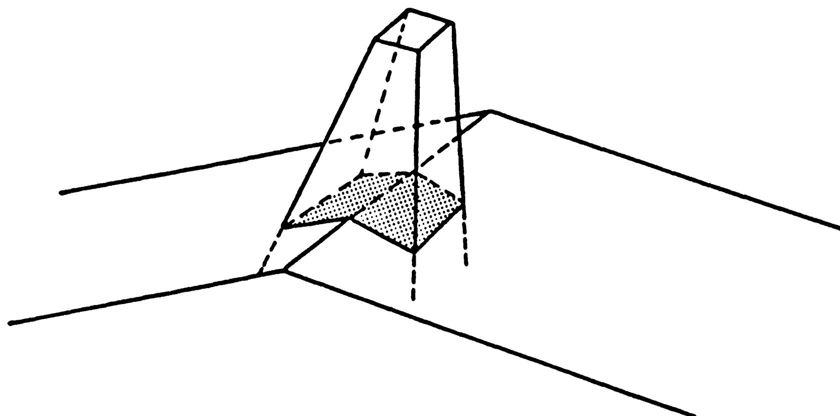


FIG. 3.2. *Co-region of R (dotted area).*

DEFINITIONS. A *lace* is a connected component of $\partial P \cap \partial Q$. A *region* is a connected component of $\partial(P \cup Q) \setminus (\partial P \cap \partial Q)$. A *co-region* is a connected component of $\partial(P \cap Q) \setminus (\partial P \cap \partial Q)$.

We are now ready to define the notion of broadcasting. A *broadcaster* is an algorithm that takes as input a vertex v on a lace and a color c and returns at least one vertex on each of the laces bounding the unique c-colored region incident upon v. Each vertex of a lace is determined by the intersection of a facet of P (or Q) and an edge of Q (or P): It is understood that this correspondence should be provided in full by the broadcaster. Suppose without loss of generality that the broadcaster is given the color red as input. Then one solution for the broadcaster is to traverse the relevant co-region on the boundary of P and keep track of the current location in the primal shield of Q until all the desired laces have been reached. In that case we say that the broadcasting is *anchored* to P. As we shall see, there is another, slightly more complicated solution, which is to stay anchored to Q and traverse the relevant portion of its boundary while tracing the navigation in the dual shield of P. In all cases the term *anchor* refers to the polytope on whose boundary the traversal takes place, the other polytope providing the guiding shield.

LEMMA 3.1. *If a broadcaster is available, it is sufficient to know a vertex of one lace of Σ (or to know that there is no lace) in order to compute the entire intersection of P and Q while incurring only linear overhead on top of the broadcasting time.*

Proof. If Σ has no lace, just knowing this fact will be enough for us to compute $P \cap Q$ in linear time, by checking whether a vertex v of P lies inside or outside Q.

Indeed, $P \cap Q = P$ (respectively, $P \cap Q = Q$) if and only if $v \in Q$ (respectively, $v \notin Q$). Suppose now that Σ has at least one lace. They key observation is that if the regions of Σ are to be made into the nodes of a graph, with arcs connecting nodes whose associated regions are bounded by a common lace, then the graph in question will be connected. Therefore, starting from the vertex given to us by the input, a standard graph traversal algorithm will allow us to discover a vertex for each lace of Σ. Since the facets of P and Q are triangles, it is elementary to compute an entire lace in time linear in its size by tracing it from one of its vertices. Once we know all the laces in full, we can easily compute $P \cap Q$ in linear time by marking the laces in both ∂P and ∂Q and exploring the facets of the co-regions. □

B. AN INTERSECTION ALGORITHM BASED ON CO-ANCHORED NAVIGATION. To illustrate the utility of broadcasting further and bring out some of its main features, we describe an $O(n \log n)$ time algorithm for intersecting P and Q that relies on a symmetric form of broadcasting, one where the anchoring alternates between P and Q (hence the term *co-anchored*). The idea is to stay glued to the boundary of $P \cap Q$ and confine our traversals to co-regions. In one broadcast we are anchored to P; that is, we navigate across the primal shield of Q while exploring a co-region in ∂P. The next time around, roles are reversed and we find ourselves broadcasting across the primal shield of P while being anchored to Q. Dual shields are never used. For this reason co-anchored navigation relies solely on what we call *primal broadcasting.*

Co-anchored navigation is simple, but it leaves us little room for improvement. Later we will develop a more complex but more promising form of navigation in which we always remain anchored to the same polytope. This requires keeping track of where we are within the primal shield of Q while we lie inside Q (primal broadcasting) and where we are in the dual shield of Q when we navigate outside Q (dual broadcasting). This also necessitates the use of mutations, the transition operation we mentioned earlier. Briefly, the reason why this more complicated form of navigation is preferable is that by always remaining anchored to the same polytope, we remain free to choose the anchor, whereas, before, the choice of anchors was dictated by the color of the co-regions. This added freedom is the key to being able to balance costs between the two polytopes and achieve linear time.

Let us now describe the simpler, co-anchored form of navigation that, as we said, relies exclusively on primal broadcasting. Given a vertex v of a certain lace γ of Σ, let R be the red region of Σ incident upon v and let R^c be its co-region. Suppose now that the broadcaster is given the vertex v and the color red as input. Its goal is to find vertices on all the laces $\gamma, \gamma_1, \cdots, \gamma_l$ of R. First of all, we (the broadcaster) can easily compute the entire lace γ by tracing the connected component of $\partial P \cap \partial Q$ passing through v. Since the boundaries are triangulated, the computation is linear in the size of γ.

Now we must reach out to all the other laces γ_i. Our strategy relies on the fact that the closure of a co-region is an *edge-connected* bounded surface (meaning that the vertices and edges form a connected graph). This seemingly obvious fact should not be taken for granted, because it does not necessarily hold for the closure of a region. In Fig. 3.3, for example, the dotted area represents a region facet that is biconnected and therefore is not edge-connected. So we must prove our claim. Suppose that two vertices of Σ in the closure of a co-region R^c cannot be joined by a path of edges in cl R^c. Then, either one of these vertices can be separated from the other by a simple closed curve that lies entirely in cl R^c but does not cut across any edge or vertex. It easily follows that this curve must lie in a single facet f of cl R^c and that f is perforated. A local analysis of the perforation reveals that f contains two points p

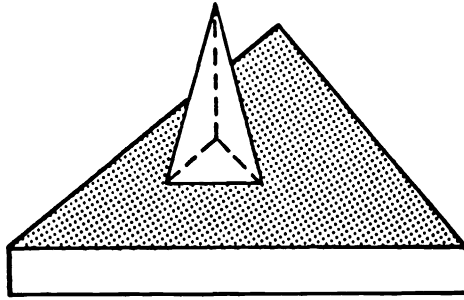


FIG. 3.3. *Region facet (dotted area) that is biconnected and therefore is not edge-connected.*

and q such that $pq \not\subseteq P \cap Q$. This contradicts the convexity of $P \cap Q$ and proves our claim.

Observe that cl R^c has vertices in $\partial P \cap \partial Q$ as well as possibly in $\partial P \backslash \partial Q$ (we previously assumed that R is red). The main difference is that the latter vertices are known ahead of time, whereas the former (the lace vertices) are discovered during the broadcast. Using standard graph traversal techniques, we can reach the vertices of all the γ_i's from v. The only problem is that whenever we visit an edge of P that crosses the boundary of Q, we need to know about it. The solution is to keep track—at all times during the broadcast—of where we are in the primal shield of Q. This means computing the intersections of the visited edges of P with the facets of the primal shield of Q. It follows trivially from Lemmas 2.1 and 2.2 that the broadcast will take $O(\rho \log n)$ time, where ρ denotes the number of edges in cl R^c. Obviously, we shall exchange the roles of P and Q if the color blue is given as input to the broadcaster. To summarize, the cost of all the broadcasts will be proportional, up to a logarithmic factor, to the size of all the co-regions. This gives us a total broadcasting time of $O(n \log n)$.

We will now be in a position to apply Lemma 3.1 and compute the entire intersection of P and Q as soon as we know one lace vertex. To do this, we take an arbitrary *starting vertex* v of P and check whether it lies inside Q. If the answer is yes, we pursue the search and locate v in the primal shield of Q. From there, we start a regular broadcast-like routine, which involves traversing ∂P and keeping track of where we are in the primal shield of Q at all times. Either we will reach the boundary of Q and, hence, a vertex of a lace, or we won't. In the latter case, we know that $P \cap Q = P$. If v lies outside Q, on the other hand, we locate the point $w = Ov \cap \partial Q$ in the primal shield of P. Let z be one of the vertices incident upon the unique (simplicial) face of Q that contains w. Let us traverse the primal shield of P along the oriented segment wz. If we do not exit P (or if $w = z$) we are just back to the previous case, with the roles of P and Q reversed. If we do leave P, however, the exit point belongs to a lace and therefore is a valid starting vertex (or it lies on an edge incident upon one). In view of Lemmas 2.1, 2.2, and 3.1, we now have an intersection algorithm with $O(n \log n)$ running time.

C. A TOPOLOGICAL EXCURSION. Before we can switch to single-anchored navigation and describe dual broadcasting, we must further explicate the relationship between polytopes and their duals. This section introduces the notions of *belts*, *bracelets*, and *co-dual regions*, on which dual broadcasting is founded. This introduction might be a little tedious, but it is indispensable for a proper understanding of why the intersection algorithm actually works.

Everything we said of P and Q (e.g., laces, regions, co-regions) applies just the same to P^δ and Q^δ. In the following, Σ^δ will designate the analog of Σ vis-à-vis $P^\delta \cup Q^\delta$; that is, $\Sigma^\delta = \partial(P^\delta \cup Q^\delta)$. We assume that the boundaries of all four polytopes P, P^δ, Q, Q^δ have been triangulated in accordance with their shields (and, hence, may have coplanar facets). Correspondence between a polytope X and its dual is ensured by the usual pointers between a k-face and its dual $(2-k)$-face. This concerns the state of faces prior to boundary triangulation. With the introduction of simplicial faces, however, this representation must be slightly amended. Let us distinguish between the *old* faces of X (before triangulation of ∂X) and the *new* ones. Note that some of them, vertices in particular, are both old and new. Each vertex of X points to any one of the new facets to which it is dual, and each new facet points to its unique dual vertex. Each old edge of X points to the unique old edge of X^δ that is dual to it. Each new edge e points to the four old edges of X that are both adjacent to e and incident upon the old facet where e lies. It is a simple exercise to set up all these pointers in linear time. An attractive aspect of this representation is that, given a new facet abc of X, we can gain constant time access not only to its dual vertex v, but also to three new facets of X^δ that are dual to a, b, and c, respectively, and incident upon v (Fig. 3.4).

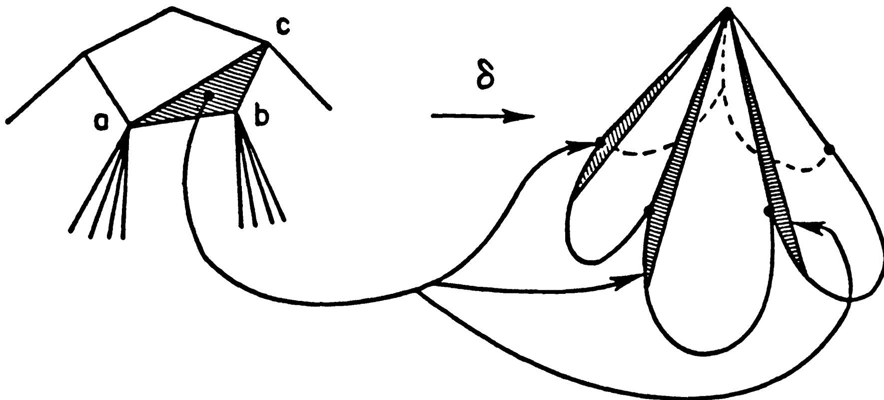


FIG 3.4. *Relationship between polytopes and their duals.*

Belts. We will show that the laces of Σ can be individually "covered" by disjoint *belts* that dualize to laces of Σ^δ. Let us color yellow all the faces of the convex hull H of $P \cup Q$ that are faces of neither P nor Q. A maximal connected subset of yellow faces is a polyhedron (in our definition), which we call a *belt* of Σ. Formally, we define *belts* and *co-belts* as follows:

DEFINITIONS. Let H be the convex hull of $P \cup Q$. A *belt* is a connected component of $(\partial H)\backslash\partial(P \cup Q)$. A *co-belt* is a connected component of $\partial(P \cup Q)\backslash\partial H$.

Note that belts and co-belts are relatively open. To be able to say more about them, we define the *envelope* of a polyhedron X (in a slightly nonstandard manner) as the set of planes π such that (i) the affine closure of $X \cap \pi$ has dimension at least 1 and (ii) X lies entirely in either one of the closed halfspaces defined by π. It easily follows from the incidence-preserving properties of a polarity that the envelope of a belt B dualizes to a lace B^δ of Σ^δ. More specifically, as we walk along the lace B^δ, a certain plane $\pi(x)$ *rolls* around the belt in a continuous fashion as x goes around S^1. Therefore, B is a simple cycle of simplicial facets and edges (and no vertices), $t_0, e_0, \cdots, t_m, e_m$, where each facet t_i is incident upon the two edges e_i and e_{i-1}

(mod $m+1$). It follows that a belt is topologically equivalent to an open annulus $S^1 \times (0, 1)$. Its frontier consists of two monochromatic connected components: One of them, denoted b_c, is blue, and the other, b_m, is red. Let us look at these components more closely. We may restrict ourselves to one of them, say b_c.

In general, b_c will be a simple closed polygonal curve, but unfortunately this might not always be the case.[1] Indeed, consider an old (i.e., untriangulated) facet f of P^δ that contributes at least one edge to the lace B^δ. It is certainly possible for B^δ to come in and out of f repeatedly, as is shown in Fig. 3.5 (where f is the horizontal face of the lower polyhedron). To translate this into the language of belts, let $u_0, \cdots, u_k$ be the cycle of edges of b_c encountered while visiting $t_0, \cdots, t_m$ in that order. Identifications of the form $u_i = u_j$ might occur. The topological type of a belt enforces two important restrictions on the allowable identification patterns. First, three or more edges cannot be identified together. Second, in any subcycle u_i, u_j, u_k, u_l we cannot have both $u_i = u_k$ and $u_j = u_l$, which means that the identifications form a parenthesis system. Of course, we might have vertex identifications only and no edge identifications at all. It might even be the case that b_c consists of a single vertex, which will happen if B^δ lies entirely in a single facet of P^δ—see the top vertex in Fig. 3.6. The only reason that b_c is not always topologically equivalent to S^1 is that the boundaries of P and Q are not smooth. If they were, then b_c would actually be diffeomorphic to S^1. Thus, if we look at ∂P and ∂Q as limit sets of smooth 2-manifolds, it appears immediately that b_c is a simple closed curve that may have been pinched and collapsed along vertices and edges according to a parenthesis system. A traversal of such a curve is

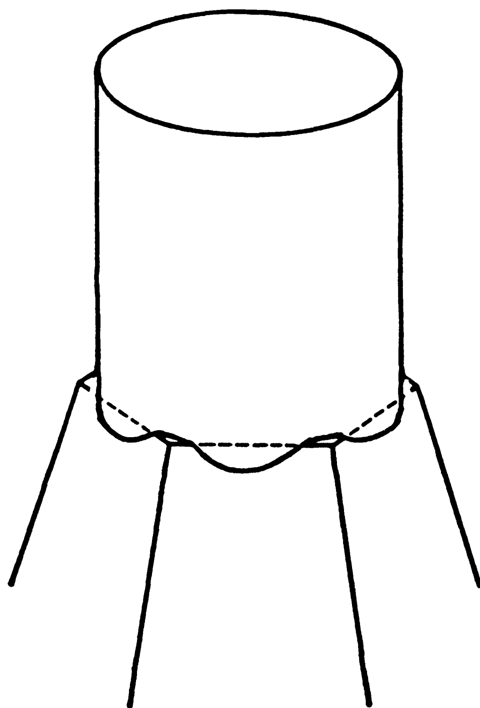


FIG. 3.5. *B^δ may come in and out of f repeatedly.*

[1] A similar situation occurs in the merge step of Preparata and Hong's convex hull algorithm [24], which is also discussed in detail in Edelsbrunner [11].

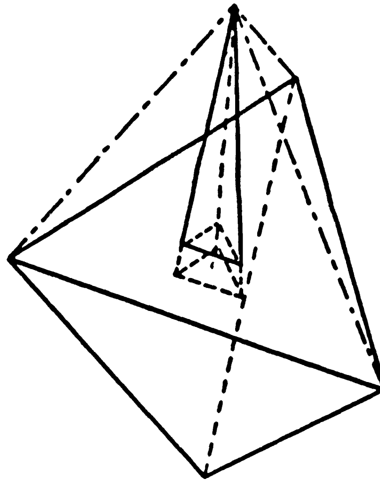


FIG. 3.6. *A case in which b_c consists of a single vertex.*

shown in Fig. 3.7: The curve can be obtained from a circle by pinching it and gluing it at various places.

Observe that two distinct laces cannot share vertices or edges, but they can pass through the same (old) facet of P or Q. This implies that although two belts cannot share a common edge or facet, their frontiers might have vertices and edges in common (recall that a belt does not contain its frontier). This fact will require that special measures be added to the intersection algorithm.

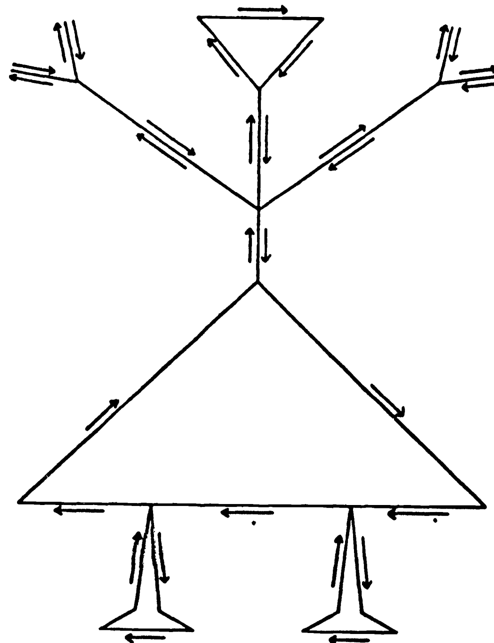


FIG. 3.7. *Traversal of a simple closed curve that has been pinched and collapsed along vertices and edges according to a parenthesis system.*

Bracelets. The space between the boundary of the convex hull H and the polytopes P and Q consists of disjoint donut-like objects, which are called *bracelets.*

DEFINITION. Let H be the convex hull of $P \cup Q$. A *bracelet* is the closure of a connected component of $H \setminus (P \cup Q)$.

A different perspective on belts and bracelets comes from looking at Σ and ∂H as the graphs of two continuous functions, respectively, φ and $\psi : S^2 \mapsto \mathbb{R}^+$. Removing the kernel of $\psi - \varphi$ from S^2 leaves relatively open connected components $S_1, S_2, \cdots$, and each belt B is the graph of ψ's restriction to some distinct domain S_i. By analogy, the graph of φ's restriction to S_i is the co-belt B^c of B. Belts and co-belts are homeomorphic and have the same frontiers. Therefore, the closure of $B \cup B^c$ is the frontier of a compact polyhedron $\mathcal{B}$, which is a bracelet of Σ: Its interior is homeomorphic to an open filled torus $D^2 \times S^1$. A bracelet is the closure of a connected component of $H \setminus (P \cup Q)$, and its belt is the relative interior of the intersection of that component with ∂H. One should not hastily conclude that a bracelet is always topologically equivalent to a filled torus. It can actually assume rather contrived shapes, because as the frontiers b_c and b_m might cause (homeomorphically) multiple point identifications along nonnull homologous paths on the torus. This might give us a filled torus pinched at various places or, as in Fig. 3.6, a closed 3-ball pinched at a pair of antipodal points. Note that general position alone cannot prevent this type of pathology.

Again, let b_c and b_m be the frontier components of B. Since Σ is locally blue (respectively, red) around b_c (respectively, b_m), we can define the maximal blue (respectively, red) connected subset B_c (respectively, B_m) of B^c whose closure contains b_c (respectively, b_m). We claim that B_c and B_m are joined together along a single lace. To prove this claim, let $\gamma = B^c \setminus (B_c \cup B_m)$ and suppose, by contradiction, that γ contains a point p of a color different from purple, say, red. Let C be the maximal connected red subset of B^c that contains p (the dotted region containing p in Fig. 3.8). The closure of C does not intersect b_c or b_m; therefore, C is a full region of Σ (and not just the portion of a region that lies within B^c). Figure 3.8 shows Σ with the white area denoting B^c and with C in the middle: It is bordered by red (i.e., dotted) material on one side and blue (i.e., hatched) material on the other. Like every region, C contains a point in ∂H. To see why, consider a plane π supporting a facet of its co-region, and let π^+ be the open halfspace bounded by π that does not contain the origin. Since P lies entirely outside π^+, we have $\pi^+ \cap \partial Q \subseteq C$; therefore, the point of $\pi^+ \cap \partial Q$ farthest away from π is a vertex of Q in $C \cap \partial H$. But this contradicts our previous observation that the portion of the bracelet $\mathcal{B}$ that lies in ∂H is confined to the closure of its belt. Consequently, γ is a purple curve whose removal from B^c creates two monochromatic surfaces of opposite color, each homeomorphic to an open annulus. It follows that γ is homeomorphic to S^1 and therefore is a lace of Σ. Thus, we have shown that $\partial\mathcal{B}$ is the disjoint union of B, b_c, B_c, γ, B_m, and b_m. In general, the removal of any one of these six sets makes $\partial\mathcal{B}$ homeomorphic to an annulus (open for the lower-case sets, closed for the upper-case ones). But one should not count on it. Removing the belt or the co-belt makes $\partial\mathcal{B}$ equivalent to a closed 2-disk with usually one open perforation, but with possibly zero (Fig. 3.6) or an arbitrarily large number of them.

Dual bracelets and co-dual regions. Each belt of Σ is associated with a unique lace of Σ. Since this is true in dual space as well, this association is bijective. Therefore, the envelope of a belt B of any bracelet $\mathcal{B}$ of Σ dualizes to the lace γ^* of a bracelet $\mathcal{B}^\delta$ of Σ^δ, whose belt B^* has for an envelope the dual of the lace γ of $\mathcal{B}$. If λ and β map a bracelet to its lace and belt, respectively, we can extend the commutative diagram

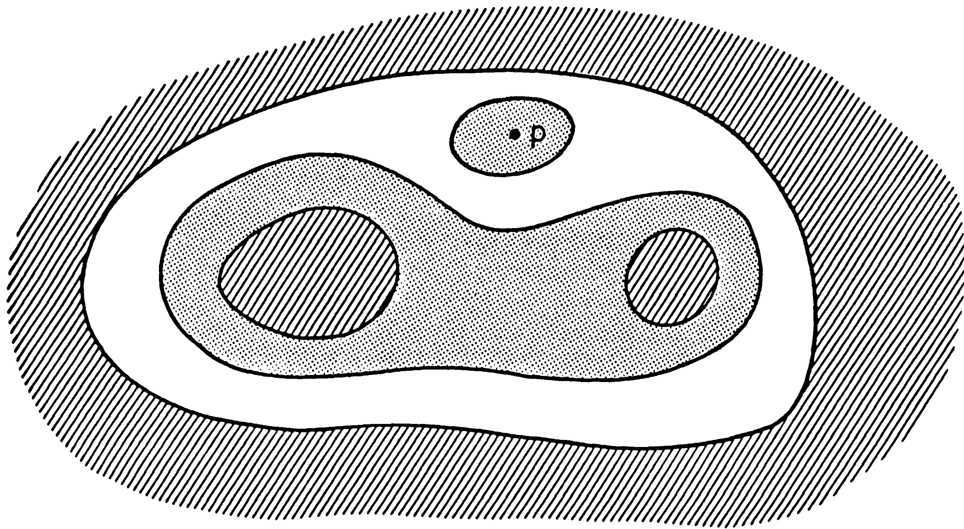


FIG. 3.8. *The maximal connected red subset of B^c that contains p (dotted region containing p) is a full region.*

of § 2.A as follows:

$$\begin{array}{ccc}
\gamma & \xrightarrow{\delta} & B^* \\
\big\downarrow{\scriptstyle \lambda^{-1}} & & \big\downarrow{\scriptstyle \beta^{-1}} \\
\mathcal{B} & \xrightarrow{\delta} & \mathcal{B}^\delta \\
\big\downarrow{\scriptstyle \beta} & & \big\downarrow{\scriptstyle \lambda} \\
B & \xrightarrow{\delta} & \gamma^*
\end{array}$$

By carefully rolling a plane around the boundary of $\mathcal{B}$ in the appropriate manner, we trace the entire boundary of $\mathcal{B}^\delta$ in dual space. (The rolling has to be "appropriate" in the sense that the passage from a belt to the co-belt and the passage across the lace must cut through the bracelet instead of tracing its envelope.) Obviously, the association between $\mathcal{B}$ and $\mathcal{B}^\delta$ is involutory. For all these good reasons, we call $\mathcal{B}^\delta$ the *dual bracelet* of $\mathcal{B}$. Note that a lace alone does not provide sufficient information to reconstruct its associated belt in dual space. One needs to add the planes supporting the facets incident to it. Another subtle point is that although the envelope of a belt in primal space is dual to a lace in dual space, the facial structures of these objects may not be in bijection. The reason is that all facets have been triangulated. As a result, a belt in primal (respectively, dual) space might end up being facially less "refined" than its corresponding lace in dual (respectively, primal) space. This will complicate our algorithm a little because of the ensuing loss of information incurred when switching to dual space.

We are now in a position to establish the most useful connection between the input polytopes and their duals, namely, a bijection between the regions of Σ and those of Σ^δ. This can be seen as a further extension of the commutative diagram. As usual, we must use caution, however, because of the nonsmoothness of polyhedral boundaries. Every bounding lace of a region is covered by a band (a co-belt) whose frontier usually consists of two simple closed curves: Figure 3.9 shows four laces (the thick black rings) surrounded by their co-belts. As we saw earlier, things might not be

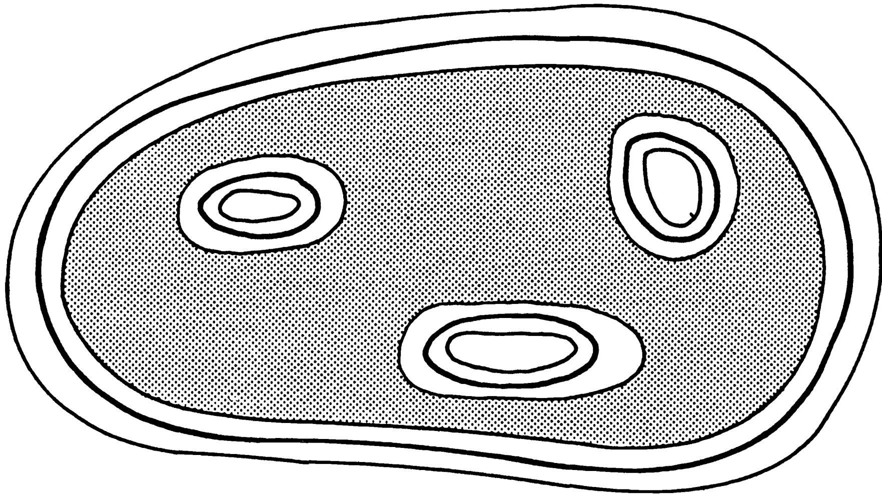


FIG. 3.9. *Four laces (thick black rings) surrounded by their co-belts.*

nearly as simple. Let R be a blue region of Σ and let $\mathscr{B}_1, \cdots, \mathscr{B}_l$ be the bracelets of its bounding laces. Since the corresponding co-belts $B_1^c, \cdots, B_l^c$ are pairwise disjoint, the set $K = R \backslash \bigcup_{1 \leqq i \leqq l} B_i^c$ is a connected subset of ∂H—the dotted area in Fig. 3.9. If again we think of P and Q as limit sets of infinite sequences of isotopic smooth manifolds, we can interpret K as a deformation retract of a copy of cl D^2 with zero, one, or several open perforations (Fig. 3.10). Thus, any two planes supporting H at points of K can be brought together by continuous rolling around H without ever leaving contact with K. This proves that among the laces of the dual bracelets $\mathscr{B}_1^\delta, \cdots, \mathscr{B}_l^\delta$, any two can be connected by a blue path in a co-region of Σ^δ. An immediate consequence is that the laces of $\mathscr{B}_1^\delta, \cdots, \mathscr{B}_l^\delta$ bound a common red region of Σ^δ. Since the argument is involutory, the region in question must be entirely bounded by these laces. For this reason it is only natural to call it the *co-dual region* R^δ of R (the prefix "co" is a reminder that it is really its co-region that is dual to R). Again, this map is involutory and, of course, consistent with the bijection previously defined between laces in primal and dual spaces.

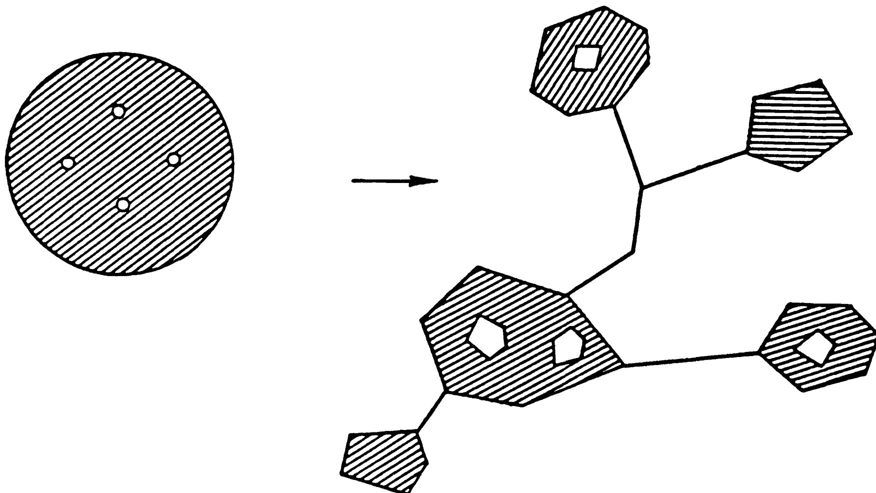


FIG. 3.10. *Interpretation of K as a deformation retract of a copy of* cl D^2 *with four open perforations.*

Unlike k-faces, which become $(2-k)$-faces in dual space, or, for that matter, laces and belts, the type "bracelet" (respectively, "region") is invariant under duality (respectively, co-duality). The best illustration of this comes from the self-dual case, where $Q=P^\delta$ (Fig. 3.6). In that remarkable situation, dual bracelets and co-dual regions remain invariant; Only their colors change! The following lemma summarizes most of our discussion so far.

LEMMA 3.2. *The interior of a bracelet $\mathcal{B}$ of Σ is homeomorphic to an open filled torus. Its boundary contains exactly one belt and one lace of Σ, which are homeomorphic to $S^1 \times (0, 1)$ and S^1, respectively. With $\mathcal{B}$ is associated a unique dual bracelet $\mathcal{B}^\delta$ of Σ^δ: The dual of the envelope of the belt of $\mathcal{B}$ is the lace of $\mathcal{B}^\delta$; conversely, the dual of the lace of $\mathcal{B}$ is the envelope of the belt of $\mathcal{B}^\delta$. Also, to each region R corresponds a unique co-dual region R^δ of Σ^δ of opposite color, and the bracelets of their bounding laces are dual to each other.*

D. DUAL BROADCASTING. As we said earlier, our goal is to break the symmetry between P and Q provided by co-anchored navigation. This will oblige the broadcaster to alternate between two modes of operation: primal and dual. The motivation for this move is to leave us the choice of anchor. Suppose once and for all that P is the anchor. (How to choose the right anchor will be discussed later.) Using the previous notation, what shall the broadcaster do if presented with the input pair (v, red)? Since we mean to let the boundary of P lead the search, this is the easy case where primal broadcasting through Q can be used (§ 3.B).

Assume now that the input is of the form (v, blue), where v is a vertex of a lace γ, and let R be the blue region to be explored. Traversing R entails leaving the polytope Q altogether. As usual, if γ is the only lace of R and we know that for a fact, everything is easy. But what if we have other laces $\gamma_1, \cdots, \gamma_l$? The difficulty is not to traverse R per se, but to tell when we might be re-entering Q and hitting upon vertices of γ_i. Primal shields are useless at this point, and we must turn to the dual shield of Q for help. Dual broadcasting from a lace of Σ to another one will be accomplished in three stages:

1. Starting from the belt of Σ^δ associated with the starting lace, navigate in dual space to the lace of its bracelet.
2. Primal broadcast through Q^δ in dual space.
3. Starting from the belt of Σ associated with a (dual space) lace newly discovered, navigate to the lace of its bracelet.

Either of the tasks performed in steps 1 and 3 is called a *mutation*: We are given a vertex on a lace of a bracelet $\mathcal{B}$, and we must find one vertex on the lace of the dual bracelet of $\mathcal{B}$. Note that from the algorithm description a mutation involves navigating from a belt to its associated lace and not the other way around. One might think that reversing the process is just dualizing it and, hence, is computationally equivalent. That is not quite true. The subtlety here relates to our previous remark about belts being facially less refined than the laces of their dual bracelets. As a result, navigating toward belts can be more difficult than toward laces (albeit still doable). We now describe an efficient implementation of a mutation. It consists of two parts: First we move to dual space, then we navigate around the boundary of the dual bracelet from somewhere on its belt to some place on its lace.

Going from the lace to its dual belt. To mutate from the lace of a bracelet $\mathcal{B}$ of Σ to the lace of its dual bracelet $\mathcal{B}^\delta$, our first action is to collect some relevant information from the input vertex v. As we indicated earlier, a vertex of a lace is never given by itself, but along with the new facet of P (or Q) and the new edge of Q (or P) upon which it is in contact. General position ensures that v is incident upon a constant

number of faces, so we can easily get two (new) facets, one in ∂P and the other in ∂Q, whose intersection contributes one edge e of the lace of $\mathcal{B}$. By duality, these two facets specify an edge ab of the belt of $\mathcal{B}^{\delta}$. Note that each facet contributes at least one of its own vertices to the bracelet: Which one can be determined in constant time by local examination. Dualizing these chosen vertices gives old facets f_a, f_b that, locally around a and b, lie on the boundary of $\mathcal{B}^{\delta}$. Note that these facets need not lie entirely in $\partial\mathcal{B}^{\delta}$. For example, in Fig. 3.11, the two (intersecting) triangles shown on the left dualize to the two vertices a and b shown on the right. The two vertices in primal space from which the arrows emanate point to their dual facets f_a, f_b, both of which extend beyond the dual bracelet. Instead of computing these old facets, which might be large, we retrieve one new facet within f_a (respectively, f_b) that is incident upon a (respectively, b). These are the hatched triangles in Fig. 3.11. Recall that we added a special provision to the correspondence between a polytope and its dual to make this possible in constant time.

Climbing down around the dual bracelet from its belt to its lace. We are now in possession of an edge ab of the belt of $\mathcal{B}^{\delta}$ and a simplicial facet A (respectively, B) incident upon a (respectively, b) that contributes a facet to $\mathcal{B}^{\delta}$ (but might not be one itself). Let P^* (respectively, Q^*) be the intersection of P^{δ} (respectively, Q^{δ}) with the plane π passing through O, a, b, and let H^* be the convex hull of P^* and Q^*. Note that, in general, H^* is *not* the intersection of π with the convex hull of $P^{\delta} \cup Q^{\delta}$. Because the origin lies in the interior of both P^* and Q^* and neither polygon contains the other, the curves ∂P^* and ∂Q^* must intersect. Furthermore, the closure $\mathcal{B}^*$ of the connected component of $H^* \backslash (P^* \cup Q^*)$ that contains the edge ab is the two-dimensional equivalent of a bracelet (Fig. 3.12): It is simple to analyze, so we will assume its basic properties. The edge ab is the "belt" of $\mathcal{B}^*$ (more appropriately called a *bridge*) and its "lace" is the point $p = \mathcal{B}^* \cap \partial\mathcal{P}^* \cap \partial\mathcal{Q}^*$. Using standard techniques, we can compute p by a simultaneous traversal of ∂P^* and ∂Q^* starting from a and b. With a bit of care, we can find p in time proportional to the number of vertices in $\mathcal{B}^*$.

Of course, this assumes that we have full knowledge of P^* and Q^*. But we do not, and we do not wish to. Since the boundaries of P^{δ} and Q^{δ} have been triangulated, however, it is easy to go from one edge to an adjacent one in constant time and thus achieve the same effect. To obtain the starting edge may require a little extra work, since A (respectively, B) might not intersect π (recall that a facet does not contain its incident vertices). But we know that A (respectively, B) lies in $\partial\mathcal{B}^{\delta}$ locally around a

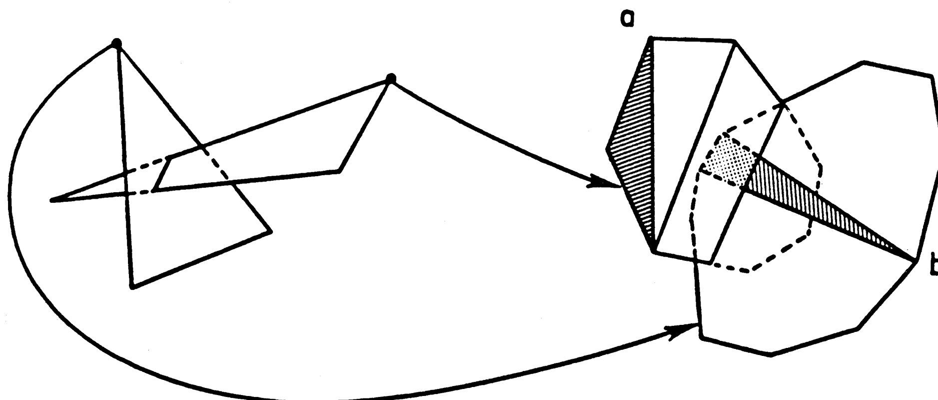


FIG. 3.11. *Two intersecting triangles (left) dualize to vertices a and b (right).*

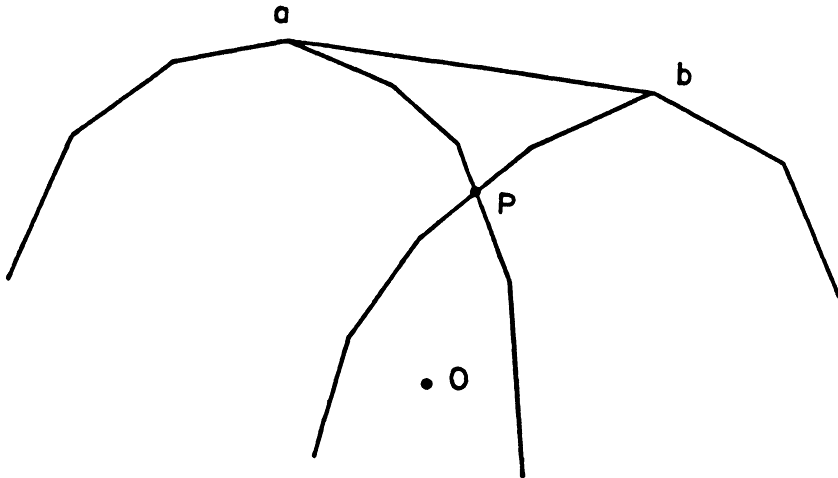


FIG. 3.12. *Two-dimensional equivalent of a bracelet.*

(respectively, b). Therefore, beginning at A (respectively, B), we can go around the cyclical order of new faces around a (respectively, b) until we find one that intersects π. If we are careful to go in the right direction, this preliminary work should involve looking only at new faces of P^δ and Q^δ that contribute to the boundary of $\mathcal{B}^\delta$. Once we have p, we also know two simplicial facets whose intersection contributes an edge to the desired lace, so the mutation is over. The total running time is at most proportional to the size of the dual bracelet.

LEMMA 3.3. *Mutating from a lace of a bracelet can be performed in time proportional to the number of edges in its dual bracelet.*

The lemma gives us all the ammunition we need to primal broadcast through Q^δ. By definition, this will reveal to us one vertex for each lace bounding the co-dual region of R. By virtue of Lemma 3.2, mutating back from each lace bounding R^δ will finally take us to the remaining laces of R and complete our dual broadcasting routine. Thus, to summarize, dual broadcasting is effected in a three-step sequence: (1) mutate to dual space, (2) primal broadcast in dual space, and (3) mutate back to primal space. One should appreciate that dual broadcasting is more than simply primal broadcasting in dual space. Another basic observation is that the input to a primal broadcast need not be a vertex of a lace: Any nonlace vertex v in a co-region of Σ (respectively, Σ^δ) will work just as well, as long as we know the location of v in the primal (respectively, dual) shield of Q. This might be handy when looking for a starting vertex.

What is the cost of broadcasting as a whole? Let κ (respectively, κ') be the maximum number of nested polytopes in the primal (respectively, dual) shield of Q whose boundaries are cut by a single (old or new) edge of P (respectively, P^δ). It follows from Lemma 2.2 that the broadcasting time will be $O((\kappa+\kappa'+1)n)$, not counting mutations. But from Lemma 3.3 the cost of all mutations is at most proportional to the number of edges in Σ and Σ^δ, which is $O(n)$. In light of Lemmas 2.1 and 3.1, we can draw the following conclusions:

LEMMA 3.4. *Given a starting vertex, we can compute the intersection of P and Q in time $O(n+\kappa n+\kappa' n)$, where κ (respectively, κ') is the maximum number of nested polytopes in the primal (respectively, dual) shield of Q whose boundaries are cut by a single edge of P (respectively, P^δ).*

What is a valid starting vertex in the context of the lemma? Any vertex on a lace of Σ or Σ^δ will do, as will any vertex of P (respectively, P^δ) that lies in the primal

(respectively, dual) shield of Q and has been located in it. It is not difficult to compute a starting vertex in linear time. Therefore, we could package our findings into yet another $O(n \log n)$ algorithm for intersecting two convex polyhedra of size n. But we can do better than that. To begin with, we must trade our full shields for k-shields.

E. USING PARTIAL SHIELDS. The only data structure we shall need is the k-shield of Q, where k is a fixed constant to be determined later. Let $Q = Q_0 \supset Q_1 \supset \cdots \supset Q_k$ and $Q^\delta = Q'_0 \supset Q'_1 \supset \cdots \supset Q'_k$ be the sequences of nested polytopes provided by, respectively, the primal and dual parts of the k-shield. Suppose that the intersections $P \cap Q_k$ and $P^\delta \cap Q'_k$ are fully available. We will show that we can emulate primal broadcasting through Q and Q^δ, even though we have only a portion of the shield at our disposal. Let us discuss the case of P and Q_k, with the understanding that the same applies to P^δ and Q'_k. Assume that both the boundaries and the interiors of P and Q_k intersect. Then, the entire theory of regions, co-regions, laces, belts, and bracelets applies verbatim to the surface $\partial(P \cup Q_k)$. Since the intersection of P and Q_k is available, we can *precompute* all primal broadcasting through Q_k anchored to P. We do this by marking the (new) facets of P and Q_k that contribute an edge to a lace of $\partial(P \cup Q_k)$. Also, for each region of $\partial(P \cup Q_k)$, we link together representative vertices of its bounding laces into a circular list. In this way, we are able to primal broadcast from any vertex of a lace in $\partial(P \cup Q_k)$ by tracing the lace in question until we hit a representative vertex. From there, we jump to all the other laces bounding the desired region in time proportional to their number. This gives us the capability to primal broadcast through the polytope Q with P as anchor, even though we might know only the outer layers of its primal shield. Obviously, the same trick can be used for primal broadcasting in dual space. Note that mutations are not affected by this change. The advantage of this new scenario is to place an upper bound of k on the values of κ and κ' in Lemma 3.4.

What now qualifies as a starting vertex? As usual, any vertex on a lace of Σ or Σ^δ will do. Another valid situation is a vertex of P (respectively, P^δ) that lies in the (possibly disconnected) set $Q \backslash Q_k$ (respectively, $Q^\delta \backslash Q'_k$), along with its location in the primal (respectively, dual) part of the k-shield of Q. Finally, any intersection between ∂Q_k (respectively, $\partial Q'_k$) and an edge of P (respectively, P^δ) and its location in the k-shield would form an appropriate starting vertex. Note that we do not extend this qualification to just any point in $P \cap \partial Q_k$ or $P^\delta \cap \partial Q'_k$ because these points might not be reached by any broadcast. Indeed, primal broadcasting anchored to P involves traversing the edges of P and *not* its facets. Thus, there could be a nonempty intersection between ∂P and ∂Q_k, even though no edge of P intersects Q_k. In that case, the primal broadcast would never make contact with Q_k, so, obviously, no point of ∂Q_k should serve as a valid starting vertex.

The intersection algorithm.

1. Check whether the interiors of P and Q intersect and conclude immediately if they do not, using the information provided by the Dobkin–Kirkpatrick algorithm. Else, pick a point O in the interior of both P and Q, and compute their dual polytopes P^δ and Q^δ.

2. Unless the anchor has already been chosen, declare P the anchor and compute the k-shield of Q. Let Q_k (respectively, Q'_k) be the innermost polytope in the primal (respectively, dual) part of the k-shield.

3. Compute $P \cap Q_k$ and $P^\delta \cap Q'_k$ recursively (the boundary case where any of the polytopes involved has constant size can be handled directly in linear time). *Crucial point*: Make Q_k and Q'_k the anchors in the recursive calls.

4. If $P \cap Q_k = P$, return P as the intersection of P and Q, and stop. If $P^\delta \cap Q'_k = P^\delta$, return Q as the intersection of P and Q, and stop.

5. If the interiors and boundaries of P (respectively, P^δ) and Q_k (respectively, Q'_k) intersect, then precompute all primal broadcasting through Q_k (respectively, Q'_k) anchored to P (respectively, P^δ).

6. Compute a starting vertex (see below).

7. Launch a broadcast from the starting vertex and pursue it until all the laces of $\partial(P \cup Q)$ have been found.

8. Use the laces to compute $P \cap Q$ explicitly.

A few comments about the algorithm are in order. Step 1 uses the linear time algorithm of Dobkin and Kirkpatrick [7]. If there is no intersection, the algorithm will say so and report the two closest points in P and Q. If P and Q intersect only at their boundaries (which, incidentally, is against our general position assumption), the Dobkin-Kirkpatrick method will still allow us to compute the full intersection in linear time. If we have a full-fledged intersection, however, the method will return a point interior to both polytopes. The dual polytopes of P and Q are easily computed in linear time. In step 2, we declare either one of the two polytopes, say, P, the anchor, unless we are responding to a recursive call, in which case the choice of anchor is forced upon us. From Lemma 2.1, the k-shield of Q can be computed in linear time. Step 3 consists of two recursive calls. As the analysis will show, switching anchors is a crucial feature of the algorithm. Failure to do so would jeopardize the linearity of the algorithm. Step 4 takes care of two trivial terminating cases. In step 5, we build the shortcuts, if any, provided by $P \cap Q_k$ and $P^\delta \cap Q'_k$.

Step 6 determines a starting vertex. If the boundaries of P and Q_k intersect, we pick a starting vertex among the vertices of $\partial P \cap \partial Q_k$ that emanate from an edge of P (if any). Then we locate the vertex in question in the primal part of the k-shield of Q. If this does not work, we try the same operation in dual space. Namely, if the boundaries of P^δ and Q'_k intersect, we pick a starting vertex among the vertices of $\partial P^\delta \cap \partial Q'_k$ that emanate from an edge of P^δ (if any). Then we locate the vertex in question in the dual part of the k-shield of Q. If this also fails, then because of step 4 we know that the skeleton (i.e., set of vertices and edges) of P (respectively, P^δ) lies entirely outside Q_k (respectively, Q'_k). Therefore, we pick a vertex v of P and check whether it lies in Q. If it does, then it must be sandwiched between Q and Q_k, so we can locate v in the primal part of the k-shield and make it the starting vertex. Otherwise, let f be a facet of P incident upon v. If the plane passing through f does not intersect Q, then its dual is a vertex of P^δ in $Q^\delta \backslash Q'_k$ and therefore qualifies as a starting vertex. Otherwise, computing the intersection allows us to identify a point w in $\partial Q \backslash P$ or $\partial P \cap \partial Q$. The cross section of P and Q by the plane passing through O, v, w consists of two convex polygons whose boundaries intersect. Any boundary intersection qualifies as a starting vertex. So, in all cases, finding a starting vertex (step 6) takes linear time. With such a vertex in hand, Lemma 3.4 tells us how to compute the intersection of P and Q. Figure 3.13 attempts to illustrate the main phases of the algorithm in two dimensions. The polytope P is the nonconvex (sorry about that) blob wiggling across the primal part of the k-shield of Q (which itself, obviously, should not be made of *disjoint* rings $\cdots$).

Complexity analysis. Put $m = p + q$, where p (respectively, q) is the total number of vertices and bounding planes in P (respectively, Q), and let $T(p, q)$ be the worst-case running time of the algorithm. If either $p = O(1)$ or $q = O(1)$, then, trivially, $T(p, q) = O(p+q)$. From Lemmas 2.1 and 3.4 we derive the general relation

$$T(p, q) = 2T(p, 3(1-1/7)^k q) + O(p+q).$$

This recurrence alone is rather ominous looking. However, the trick of switching anchors at each recursive call will now pay off. Indeed, the recurrence can be more

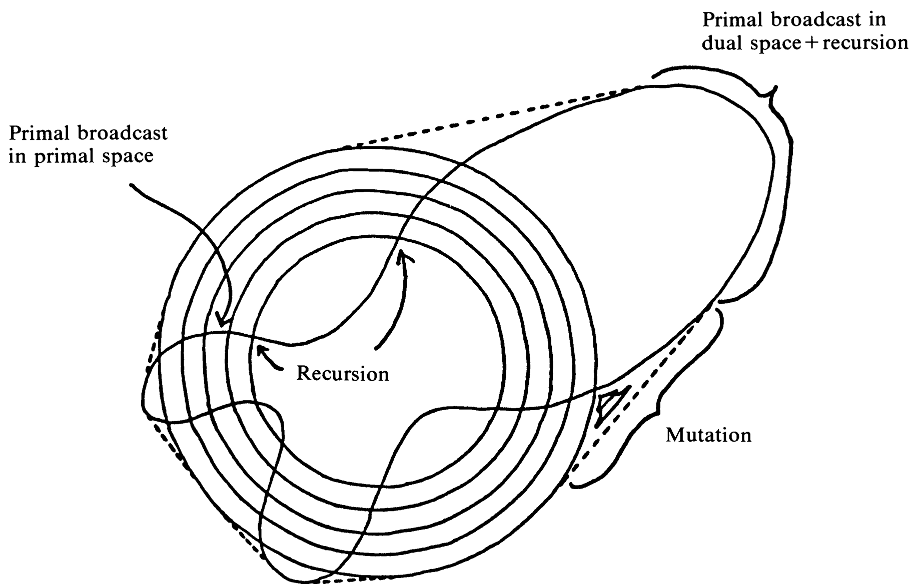


FIG. 3.13. *A bird's-eye view of the algorithm.*

accurately expressed as

$$T(p, q) = 2T'(p, 3(1-1/7)^k q) + O(p+q)$$

and

$$T'(p, q) = 2T(3(1-1/7)^k p, q) + O(p+q),$$

which, after substitution, yields

$$T(p, q) = 4T(3(1-1/7)^k p, 3(1-1/7)^k q) + O(p+q)$$

or, more simply,

$$T^*(m) = 4T^*(m/5) + O(m),$$

where $T^*(m) = T(m, m)$ and $k = 18$. We have $T^*(m) = O(m)$, and thus ends our search for a linear time algorithm for intersecting two convex polyhedra.

Reflecting back on the algorithm, it is interesting to observe that switching anchors at each recursive call makes all the difference. The process can be regarded as a form of *branching dovetailing.* A second observation is that, in the end, all the laces in both Σ and Σ^δ will have been fully computed (or at least this can be easily ensured at no extra asymptotic cost). Since the laces of Σ^δ dualize to the belts of Σ, we get the convex hull of P and Q as a bonus. Of course, another method is to compute the intersection of the dual polytopes and dualize back. If P and Q are disjoint, then we can use the Preparata–Hong linear time wrapping routine. In all cases, therefore, we are able to compute the convex hull of two convex polytopes in linear time.

THEOREM 3.5. *It is possible to compute the intersection* (*and the convex hull*) *of two three-dimensional convex polyhedra in linear time. It is assumed that the polyhedra are given in standard representation.*

4. Miscellaneous applications. The intersection algorithm can be put to use for improving or simplifying the solutions to a number of geometric problems. These applications are all very simple, so we keep our discussion to a minimum.

A. INTERSECTING SEVERAL CONVEX POLYHEDRA. Consider the problem of computing the common intersection of k convex polyhedra $P_1, \cdots, P_k$, given in standard representation. We can do this in optimal $O(n \log k)$ time, where n is the total number of vertices among the k polytopes. We use a straightforward scheme, borrowed from multi-way merging: Put the polyhedra in bijection with the leaves of a complete binary tree, and compute intersections in an order consistent with the tree. The $O(n \log k)$ running time of this algorithm is worst-case optimal even in two dimensions, because we can reduce any k-way merge to polygon intersection. To see this, consider a collection of k sorted lists $L_1, \cdots, L_k$ with distinct elements. Form the polygons $P_1, \cdots, P_k$, where P_i is the unbounded polygon defined by the intersection of the halfplanes $y \geqq x_j(2x - x_j)$: P_i is bounded by the tangents to the parabola $y = x^2$ at the points (x_j, x_j^2), for $j = 1, \cdots, m$, where $L_i = (x_1, \cdots, x_m)$. Now, observe that the boundary of the polygon $P = \bigcap_{1 \leqq i \leqq k} P_i$ contains all the points in $\{(x, x^2) \mid x \in \bigcup_{1 \leqq i \leqq k} L_i\}$. Therefore, the merged sequence of all the k lists can be read off by going around the boundary of P. To obtain the desired lower bound, we form k lists of size $m = n/k$ and observe that they can be merged in $M = \binom{n}{m_1, \cdots, m_k}$ ways, where $m_i = m$. The lower bound follows from the fact that $\log M = \Omega(n \log k)$ and by-now standard algebraic decision-tree arguments [25].

B. CONVEX HULLS. Bentley and Shamos [4] have shown how to take advantage of certain point distributions to obtain linear expected-time algorithms for computing convex hulls. The idea is to use divide-and-conquer by splitting the input set in a fixed manner (independent of the point set itself) and build the convex hull bottom-up. For their method to work efficiently, the merge step must be capable of computing the convex hull of two (possibly intersecting) convex polytopes reasonably fast. As observed by Seidel [11], we can use the Preparata–Hong algorithm for that purpose and get linear expected complexity for a wide class of point distributions. Using Theorem 3.5 widens that class. Specifically, any distribution for which the average size of the convex hull of a random set of n points is $O(n/\log^{1+\varepsilon} n)$ will trivially yield a linear expected-time complexity.

C. MERGING VORONOI DIAGRAMS. Kirkpatrick [17] has shown that two planar Voronoi diagrams can be "merged" in linear time. His algorithm is ingenious but somewhat complicated. Standard reductions cause the same result to fall straight out of Theorem 3.5. The problem is this: Given two sets of n points in the plane with their respective Voronoi diagrams, compute the diagram of their union. By using a reduction due to Edelsbrunner and Seidel [11], [14], we compute a Voronoi diagram of n points by intersecting n halfspaces. Let $h(p)$ denote the closed halfspace bounded below by the tangent to the paraboloid $z = x^2 + y^2$ at the point whose xy projection is p. The Voronoi diagram of $p_1, \cdots, p_n$ is the xy projection of the two-dimensional cell complex formed by the boundary of the convex polyhedron $\bigcap_{1 \leqq i \leqq n} h(p_i)$. Thus, merging Voronoi diagrams becomes a special case of intersecting two convex polyhedra. Applications include computing the Voronoi diagram of a polygon (Kirkpatrick [17]) and of the vertices of a convex polygon (Aggarwal et al. [1]).

5. Conclusions. Our main result is a linear-time algorithm for intersecting two convex polyhedra in 3-space. Whether the algorithm lends itself to efficient and robust implementations remains to be seen. In practice the recursion might be stopped after only a few steps, when the polytopes are small enough that we can use more naive methods. One advantage of this algorithm is that it does not use any dynamic data structure. Except for the work needed to build the shields, all other activity is purely pointer chasing through a three-dimensional cell complex. It is likely that adding randomization might lead to certain simplifications. Looking into this possibility might

be worthwhile. One must live with the fact, however, that intersecting even two tetrahedra is difficult to implement correctly, so the goal of an intersection algorithm that is truly simple to implement might be elusive. An outstanding open problem is that of intersecting two nonconvex polyhedra efficiently. The problem of intersecting arbitrarily placed triangles in 3-space has been investigated by Aronov and Sharir [2]. How much we can gain by having collections of faces structured into the boundaries of simple polyhedra is an intriguing open question.

Acknowledgments. I wish to thank Herbert Edelsbrunner, David Kirkpatrick, and Chee Yap for helpful conversations. I also thank the referees for their help in improving the presentation of the results.

REFERENCES

[1] A. AGGARWAL, L. J. GUIBAS, J. SAXE, AND P. SHOR, *A linear time algorithm for computing the Voronoi diagram of a convex polygon*, in Proc. 19th Annual ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 39-45.

[2] B. ARONOV AND M. SHARIR, *Triangles in space, or building and analyzing castles in the air*, in Proc. 4th Annual ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1988, pp. 381-391.

[3] B. G. BAUMGART, *A polyhedron representation for computer vision*, in AFIPS Conference Proceedings, Vol. 44, AFIPS Press, 1975, pp. 589-596.

[4] J. L. BENTLEY AND M. I. SHAMOS, *Divide and conquer for linear expected time*, Inform. Process. Lett., 7 (1978), pp. 87-91.

[5] B. CHAZELLE AND D. P. DOBKIN, *Intersection of convex objects in two and three dimensions*, J. ACM, 34 (1987), pp. 1-27.

[6] D. P. DOBKIN AND D. G. KIRKPATRICK, *Fast detection of polyhedral intersection*, Theoret. Comput. Sci., 27 (1983), pp. 241-253.

[7] ———, *A linear algorithm for determining the separation of convex polyhedra*, J. Algorithms, 6 (1985), pp. 381-392.

[8] D. P. DOBKIN AND M. J. LASZLO, *Primitives for the manipulation of three-dimensional subdivisions*, in Proc. 3rd Annual ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1987, pp. 86-99.

[9] D. P. DOBKIN AND J. I. MUNRO, *Efficient uses of the past*, J. Algorithms, 6 (1985), pp. 455-465.

[10] M. E. DYER, *Linear time algorithms for two and three-variable linear programs*, SIAM J. Comput., 13 (1984), pp. 31-45.

[11] H. EDELSBRUNNER, *Algorithms in Combinatorial Geometry*, Springer-Verlag, Berlin, New York, 1987.

[12] H. EDELSBRUNNER AND H. MAURER, *Finding extreme points in three dimensions and solving the post-office problem in the plane*, Inform. Process. Lett., 21 (1985), pp. 39-47.

[13] H. EDELSBRUNNER AND E. P. MÜCKE, *Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms*, in Proc. 4th Annual ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1988, pp. 118-133.

[14] H. EDELSBRUNNER AND R. SEIDEL, *Voronoi diagrams and arrangements*, Discrete Comput. Geom., 1 (1986), pp. 25-44.

[15] L. J. GUIBAS AND J. STOLFI, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams*, ACM Trans. Graphics, 4 (1985), pp. 75-123.

[16] S. HERTEL, M. MÄNTYLÄ, K. MEHLHORN, AND J. NIEVERGELT, *Space sweep solves intersection of convex polyhedra*, Acta Inform., 21 (1984), pp. 501-519.

[17] D. G. KIRKPATRICK, *Efficient computation of continuous skeletons*, in Proc. 20th Annual IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1979, pp. 18-27.

[18] ———, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28-35.

[19] N. MEGIDDO, *Linear-time algorithms for linear programming in* $\mathbb{R}^3$ *and related problems*, SIAM J. Comput., 12 (1983), pp. 759-776.

[20] K. MEHLHORN, *Data Structures and Algorithms 3: Multidimensional searching and computational geometry*, Springer-Verlag, Berlin, New York, 1984.

[21] K. MEHLHORN AND K. SIMON, *Intersecting two polyhedra one of which is convex*, University of Saarland, Tech. Report, Saarbrücken, Federal Republic of Germany, 1986.

[22] D. E. MULLER AND F. P. PREPARATA, *Finding the intersection of two convex polyhedra*, Theoret. Comput. Sci., 7 (1978), pp. 217–236.

[23] J. O'ROURKE, C. B. CHIEN, T. OLSON, AND D. NADDOR, *A new linear algorithm for intersecting convex polygons*, Comput. Graphics and Image Process., 19 (1982), pp. 384–391.

[24] F. P. PREPARATA AND S. J. HONG, *Convex hulls of finite sets of points in two and three dimensions*, Comm. ACM, 20 (1977), pp. 87–93.

[25] F. P. PREPARATA AND M. I. SHAMOS, *Computational geometry*, Springer-Verlag, Berlin, New York, 1985.

[26] C. P. ROURKE AND B. J. SANDERSON, *Introduction to Piecewise-Linear Topology*, Springer-Verlag, Berlin, New York, 1982.

[27] M. I. SHAMOS AND D. HOEY, *Geometric intersection problems*, in Proc. 17th Annual IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1976, pp. 208–215.

[28] C. K. YAP, *A geometric consistency theorem for a symbolic perturbation scheme*, in Proc. 4th Annual ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1988, pp. 134–142.

SIAM J. COMPUT.
Vol. 21, No. 4, pp. 697–712, August 1992

© 1992 Society for Industrial and Applied Mathematics
005

THE POWER OF THE QUEUE*

MING LI†, LUC LONGPRÉ‡, AND PAUL VITÁNYI§

Abstract. Queues, stacks, and tapes are basic concepts that have direct applications in compiler design and the general design of algorithms. Whereas stacks (pushdown store or last-in–first-out storage) have been thoroughly investigated and are well understood, this is much less the case for queues (first-in–first-out storage). In this paper a comprehensive study comparing queues to stacks and tapes (off-line and with a one-way input tape) is presented. The techniques used rely on Kolmogorov complexity. In particular, one queue and one tape (or stack) are incomparable:

(1) Simulating one stack (and hence one tape) by one queue requires $\Omega(n^{4/3}/\log n)$ time in both the deterministic and the nondeterministic cases. A corollary of this lower bound states that for this model of one-queue machines, nondeterministic linear time is not closed under complement.

(2) Simulating one queue by one tape requires $\Omega(n^2)$ time in the deterministic case and requires $\Omega(n^{4/3}/(\log n)^{2/3})$ in the nondeterministic case.

The paper further compares the relative power between different numbers of queues:

(3) Simulating two queues (or two tapes) by one queue requires $\Omega(n^2)$ time in the deterministic case, and $\Omega(n^2/(\log^2 n \log\log n))$ in the nondeterministic case. The deterministic bound is tight. The nondeterministic one is almost tight. The upper bounds for queues are also obtained.

Key words. abstract storage unit, multi-queue machines, multi-tape machines, on-line simulation, lower bounds, upper bounds, Kolmogorov complexity

AMS(MOS) subject classifications. 68Q05, 68Q30

1. Introduction. It has been known for over 20 years that all multi-tape Turing machines can be simulated on line by two-tape Turing machines in time $O(n \log n)$ [HS66] and by one-tape Turing machines in time $O(n^2)$. Since then, many other models of computation have been introduced and compared [Aan74], [DGPR84], [HS65], [HS66], [HU79], [KOS79], [LS81], [MSS87], [PSS81], [Pau82], [Vit85]. In addition to different storage mechanisms, real-time, on-line, and off-line machines have been studied. An on-line simulation essentially simulates step-by-step each move of the simulated machine. In this paper we consider off-line machines, for which an answer is given only after the entire input has been read. There is no need to simulate the moves of the machine; it only matters that the right answer is given. We also use the one-way input convention, which states that the machine has a one-way input tape. As usual, the machines have a finite control and access to some storage.

The relative power of stacks and tapes is more or less well known.[1] For example, for the nondeterministic case, we know that 1 stack < 1 tape < 2 stacks < 3 stacks = k

*Received by the editors July 7, 1989; accepted for publication (in revised form) July 24, 1991. Part of this paper appeared in a preliminary version in [LL V86].

†Present address, Department of Computer Science, University of Waterloo, Ontario, Canada N2L 361. Department of Computer Science, York University, North York, Ontario, Canada M3J 1P3. This research was performed while the author was at Ohio State University and Harvard University. This work was supported by the National Science Foundation under grant DCR-8606366 by the Office of Naval Research under grant N00014-85-K-0445.

‡College of Computer Science, Northeastern University, Boston, Massachusetts 02115. Part of this research was performed while the author was a visiting faculty member in the Computer Science Department at the University of Washington.

§Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, the Netherlands. The work was performed in part at the Laboratory for Computer Science, Massachusetts Institute of Technology, and supported in part by the Office of Naval Research under contract N00014-85-K-0168, by the U.S. Army Research Office under contract DAAG29-84-K-0058, by the National Science Foundation under grant DCR-83-02391, and by the Defense Advanced Research Projects Agency under contract N00014-83-K-0125.

[1]Throughout the paper, *stack* and *pushdown store* are used synonymously. The basic operations are *push* and *pop*, and only the top of the store is accessible to the machine.

stacks = k tapes, where $A < B$ means that B can simulate A in linear time, but A cannot simulate B in linear time. In most of the cases, close lower and upper bounds are known for the simulation [Maa85], [Li85b], [Li88], [LV88], [Vit84b].

In this paper we give a complete characterization of (off-line, one-way input) queue machines. The main theorems show that one-queue machines are incomparable to one-stack or one-tape machines, both deterministically and nondeterministically. One corollary of our nondeterministic lower bound is that for our model of one-queue machines, nondeterministic linear time is not closed under complement. We also compare the relative power of machines having different numbers of queues. The current knowledge of upper and lower bounds for the simulation between queues and tapes is roughly summarized in Figs. 1, 2, and 3. Figure 1 contains results that were previously known. The results of Fig. 2 are covered in §2. Notice that all the bounds in Fig. 2 are valid also for simulating one stack or two stacks. The results of Fig. 3 are covered in §3.

	deterministic	nondeterministic
upper bound	$O(n^2)$ (in [HS65])	$O(n^{3/2}\sqrt{\log n})$ (in [Li88])
lower bound	$\Omega(n^2)$ (in [LV88])	$\Omega(n^{4/3}/\log^{2/3} n)$ (in [LV88] or [Li85a])

FIG. 1. *Simulating one queue by one tape.*

	deterministic	nondeterministic
upper bound	$O(n^2)$	$O(n^2)$
lower bound	$\Omega(n^{4/3}/\log n)$	$\Omega(n^{4/3}/\log n)$

FIG. 2. *Simulating one tape, one stack, or two stacks, by one queue.*

	deterministic	nondeterministic
upper bound	$O(n^2)$	$O(n^2)$
lower bound	$\Omega(n^2)$	$\Omega(n^2/\log^2 n \log\log n)$

FIG. 3. *Simulating two queues by one queue.*

We use Kolmogorov complexity techniques [Sol64], [Kol65], [Cha77], together with some new techniques to enable us to deal with queues to prove the theorems. The Kolmogorov complexity $K(x)$ of a string x is the length of the shortest program printing the string x. By a simple counting argument, we know that for at least half of the strings x of each length, $K(x) \geq |x|$. These strings are called *incompressible* or K *random*. For completeness, we recall the notions of Kolmogorov complexity of binary strings and those of self-delimiting descriptions (see, e.g., [PSS81], [LV88]). Fix an effective coding C of all Turing machines as binary strings, such that no code is a prefix of any other code. Denote the code of Turing machine M by $C(M)$. The Kolmogorov complexity with respect to C of a binary string x, denoted $K_C(x)$, is the length of the smallest binary string $C(T)y$

such that T started on input y halts with output x. The crucial fact one uses is that for any fixed effective enumerations C and D, for all x $|K_C(x) - K_D(x)| < c$, with c a constant depending only on C and D (but not on x). Thus, up to an additive constant, the Kolmogorov complexity is independent of the particular effective enumeration chosen, which allows us to drop the subscript. With some abuse of notation, the sequel equalities and inequalities involving Kolmogorov complexity will always be assumed to hold up to an additive constant only. To be able to differentiate between parts of y such that T is able to use different parts for different purposes (can compute an r-ary function), we need the notion of self-delimiting descriptions. If $a = a_1 a_2 \cdots a_n$ is a string of 0's and 1's, then $a_1 0 a_2 0 \cdots 0 a_n 1$ is a self-delimiting description of twice the original length. More efficiently, if $b = b_1 \cdots b_m$ is the length of a in binary, then the self-delimiting description of b concatenated with a is also a self-delimiting description of a, this time of length $n + 2\log n$ instead of $2n$. For example, 1000011101 is the self-delimiting version of 1101.

2. The queue machine model. We will first describe more formally the model and the notation we use for queue machines.

A queue machine has a one-way input tape with the input head initially positioned at the beginning of the input string. For storage it uses a queue. The rear of the queue contains the first symbols pushed (and not popped). The front contains the last symbols pushed. The machine can access only one symbol at the rear of the queue.

One step of the queue machine consists of all the following. According to the old state and the contents of the cells scanned on the input and on the queue, the machine

1. reads an empty or nonempty symbol from the input,
2. pops an empty or nonempty symbol from the queue,
3. pushes an empty or nonempty symbol on the queue,
4. changes state.

Let h_{in} be the read-only head on the one-way input tape. We identify the queue with a tape with two heads h_r and h_w. The queue machine is implemented as follows on the tape representation. The initial state and the state transitions are the same. The head h_r is a read-only, one-way head on the tape. The head h_w is a write-only, one-way head on the tape. One step of the queue machine is implemented as follows:

1. the input head h_{in} behaves the same way as on the original queue machine;
2. if a nonempty symbol is written (pushed) on the queue, then h_w writes the symbol in the currently scanned cell and moves to the right adjacent cell (if an empty symbol is written, then h_w does not move);
3. if a nonempty symbol is read (popped) from the queue, then h_r moves to the right adjacent cell (if an empty symbol is read, then h_r does not move);
4. the change of state occurs as in the original machine.

Without loss of generality, we assume that the machine uses a binary alphabet on the queue and accepts by empty queue.

Let $h_k(t)$ denote the position of head $k \in \{in, r, w\}$ at time t on its respective tape. Let $c_1, c_2, \cdots, c_n$ be the individual cells on the input tape. Let $d_1, d_2, \cdots$ be the individual cells on the queue. We sometimes use $h_k(t)$ to denote the cell at that position.

The contents of the tape from $h_r(t)$ through $h_w(t) - 1$ inclusive is called the *actual queue* at time t, or *Queue*(t). The length of *Queue*(t), denoted $|$*Queue*$(t)|$, is $h_w(t) - h_r(t)$. We say that cells d_i and d_j are *contiguous* on *Queue*(t) if $h_r(t) < j < h_w(t)$ and $j = i+1$, or if $i + 1 = h_w(t)$ and $j = h_r(t)$ (that is, the cells at opposite ends of the queue are also considered contiguous).

3. Simulating one tape by one queue.

3.1. Upper bound. Our upper bound is straightforward. It is for simulating any fixed number of stacks, but since two stacks can simulate one tape in real time, our upper bound applies to tapes as well.

THEOREM 3.1. *For any fixed k, one queue can simulate k stacks in $O(n^2)$ time for both deterministic and nondeterministic machines.*

Proof. Simulate the k stacks by coding them sequentially onto the queue such that the top of each stack comes first. In front of each stack top, put a marker to indicate the separation between the stacks.

Each operation (push or pop on one stack) can be done in $O(n)$ time by scanning the entire queue and performing the local transformation after the appropriate marker. Scanning is done by successively transferring the symbols from one end of the queue to the other end. The total time is then in $O(n^2)$. This simulation can be made for deterministic or nondeterministic machines. □

3.2. Lower bound. In this section, we show that it takes $\Omega(n^{4/3}/\log n)$ time for a nondeterministic one-queue machine with a one-way input to recognize the language $L = \{w\#w^R : w \in \{0,1\}^*\}$. The proof also provides the same lower bound for the set of palindromes.

Because L can be recognized in linear time by a deterministic one-stack machine (a deterministic pushdown automaton), we can conclude that it takes $\Omega(n^{4/3}/\log n)$ time for a nondeterministic one-queue machine to simulate a deterministic one-stack machine.

The intuition behind the proof is that while the queue machine reads w, it has to store all the information in some sequential way on the queue. It turns out to be impossible to check the stored form of w for correspondence with w^R while the latter string is read from the input tape, so w^R must be stored in some sequential way as well. Using crossing sequence arguments, we show that whatever way the information is stored, the machine is forced to scan the queue many times. This repeated scanning then implies the lower bound on simulation time.

THEOREM 3.2. *A nondeterministic one-queue machine with a one-way input tape requires $\Omega(n^{4/3}/\log n)$ time to accept the language $L = \{w\#w^R : w \in \{0,1\}^*\}$.*[2]

Remark. This holds both for the worst-case time and the average time, when the average is taken over all strings in L. Notice that the straightforward algorithm to accept L with a queue has a linear average time when the average is taken over all strings, since most strings can be discovered not to be in the language quickly.

Proof. Let Q be a one-queue machine that accepts L. We show that Q will make $\Omega(n^{4/3}/\log n)$ steps before accepting any string $x\#x^R$ for incompressible strings x of size n. Since the size of the input is $2n+1$, this will provide the wanted lower bound for L. Since at least half the strings of each length are incompressible, this also provides the claimed average time lower bound.

Let x be an incompressible string of length n. We separate x into two blocks: $x = x_0\tilde{x}$, with $|x_0| = \lfloor n/2 \rfloor$. Let $m = \lfloor n^{1/3}/4 \rfloor$ and $p = \lfloor n/2m \rfloor$. We further separate $\tilde{x}$ into m blocks of size p or $p+1$: $\tilde{x} = x_1 x_2 \cdots x_m$.

[2]Here we use the stronger version of Ω where $T(n) \in \Omega(f(n))$ if there are positive constants c and n_0 such that for all $n \geq n_0$, $T(n) \geq cf(n)$. Notice that there is no string of even length in the language. To be strict, we show that the time is in $\Omega(n^{4/3}/\log n\ :\ n$ is odd). With a slightly modified language, $\{x\#x^R\} \cup \{x\#\#x^R\}$, we could prove it for all n.

We look at any fixed accepting computation of the machine on input $x\#x^R$. Let t_j be the time step when h_{in} enters the block x_j. Let t'_j be the time step when h_{in} enters the block ${x_j}^R$. If z is a substring of x, then z' denotes the corresponding substring of x^R $(= x')$.

CLAIM 3.3. *If* $t_1 \leq t \leq t'_0$, *then* $|Queue(t)| \geq n/2 - O(\log n)$.

Proof. Let $t_1 \leq t \leq t'_0$. Let $|Queue(t)| = s$. The string x can be reconstructed by using the following information: a description of this discussion and of Q in $O(1)$ bits, the string $Queue(t)$ of length s, the string $\tilde{x}$ of length $\lceil n/2 \rceil$, the state $q(t)$ of the machine in $O(1)$ bits, and $h_{in}(t)$ in $\leq \log n + 2$ bits. All items are encoded as self-delimiting strings. The total number of bits required for this description is $s + n/2 + O(\log n)$.

To reconstruct x from this information, run Q with all possible candidate strings y substituted for x_0. Single out the strings y for which there is a time step for which $Queue(t)$, $h_{in}(t)$, and $q(t)$ correspond. Among those y, the machine should accept only if $y = x_0$; otherwise, it would accept the string $x_0\tilde{x}\#\tilde{x}^R y^R \notin L$ by behaving like the computation on $x\#x^R$ up to time t and like the computation on $y\tilde{x}\#\tilde{x}^R y^R$ after time t.

Because x is incompressible, we know that $K(x) \geq n$, so it must be that our program reconstructing x has size $\geq n$. Thus, we have $s + n/2 + O(\log n) \geq n$, from which the claim follows. □

The machine Q needs to remember what it reads on the input and code it in some way on the queue or compare it with what is already on the queue. What can be written on the queue is determined by the current state, the input, and the rear of the queue. The input can be compared with the rear of the queue. These intuitive ideas motivate the following definitions of *influence*.

DEFINITION 3.4. An input cell c_i *directly influences* a cell d_j if h_{in} scans c_i while h_w writes in d_j (that is, $h_w(t) = j$, $h_w(t+1) = j+1$, and $h_{in}(t) = i$).

DEFINITION 3.5. A cell d_i *backward influences* a cell d_j if h_w is or moves onto d_i when h_r moves onto d_j (that is, $h_r(t-1) = j-1$, $h_r(t) = j$ and $h_w(t) = i$).

DEFINITION 3.6. A cell d_i *forward influences* a cell d_j if h_r scans d_i while h_w writes in d_j (that is, $h_w(t) = j$, $h_w(t+1) = j+1$ and $h_r(t) = i$).

(See Fig. 4 for an example of direct influence and Fig. 5 for an example of backward and forward influence.)

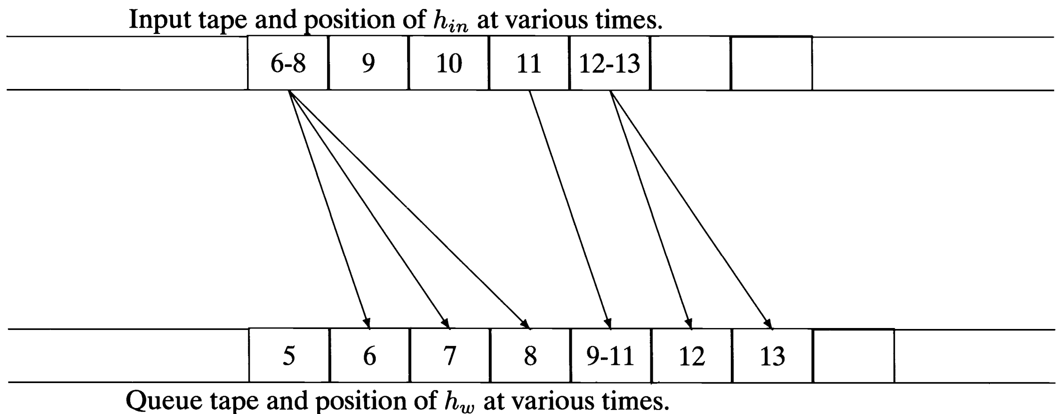


FIG. 4. *Direct influence relation.*

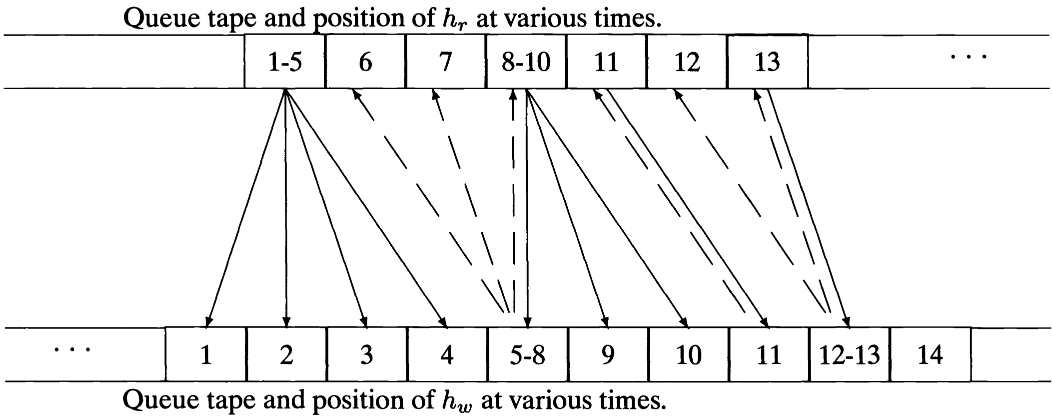


FIG. 5. *Forward (⟶) and backward (— — →) influence relation.*

DEFINITION 3.7. The *influence relation* among the tape cells is the transitive closure of the forward influence relation union the transitive closure of the backward influence relation. In other words, a cell d_i influences a cell d_j if there is a chain of forward influences or a chain of backward influences from d_i to d_j.

An input cell c_i influences a cell d_j if c_i directly influences a tape cell that influences d_j.

A block of cells influences a cell if and only if at least one of the cells in the block influences it. A block of cells is influenced by a block of cells if at least one cell of the first block is influenced by the second block. Figure 6 illustrates the concept. The influence relation will allow us to talk about where information can be stored on the queue or which information from the queue can be compared with the input.

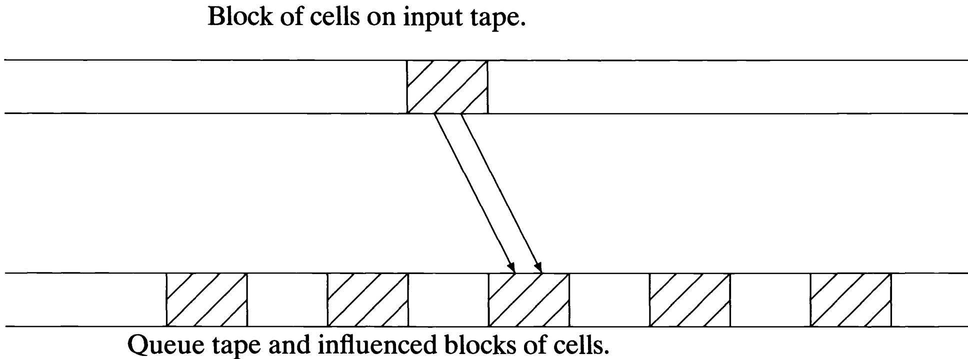


FIG. 6. *Blocks on the queue influenced by a block on the input.*

It is worth stating a few facts about the influence relations. Each tape cell is directly influenced by exactly one input cell. It is also forward and backward influenced by exactly one tape cell. The cells directly influenced by a contiguous block of input cells form a contiguous block. This holds also for forward and backward influence.

The sequence of blocks influenced by a block of input cells will be used with the crossing sequence around the blocks. Crossing sequences for queue machines need a special definition.

DEFINITION 3.8. A *partial configuration* of the machine at some time t is the state of the machine at that time, the position of all the heads on their respective tape, the contents of the cells $h_r(t)$, $h_{in}(t)$, and the contents of the cells immediately preceding those two cells.

DEFINITION 3.9. The *crossing sequence* (c.s.) associated with a cell d_i is the partial configuration at the time t when h_r goes from cell d_i to cell d_{i+1} (that is, $h_r(t-1) = d_i$ and $h_r(t) = d_{i+1}$) plus the partial configuration at the time when h_w goes from d_i to d_{i+1}. Since using more than n^2 tape cells would take too much time, we may assume that each head position can be described in $O(\log n)$ bits.

The *crossing sequence* around a region $d_i \cdots d_j$ is the c.s. associated with d_{i-1} concatenated with the one associated with d_j.

The *crossing sequence* around a list of regions is the concatenation of the c.s. around each of the regions.

Intuitively, for a deterministic computation, changing a block of input will change only the influenced regions, provided that the change does not alter the crossing sequence around the influenced regions. For a nondeterministic computation, the situation is a little more delicate, but the idea is the same. We need the backward influence to be able to deal with nondeterministic computations. A nondeterministic machine can guess the input on the queue and start the computation before the input head even moves once. A change in an input block will have "backward effects" on that computation.

For every computation path, there is a backward computation path consisting of all the configurations in reverse order. Moreover, there is a queue machine Q' that has as accepting computation paths all the backward accepting computations of Q. Just exchange the role of the read and write heads: $h'_w(t) = h_r(t)$ and $h'_r(t) = h_w(t)$. For the computation, the time and the heads go backwards. The influence definition was designed such that the forward influence on the tape for Q corresponds to the backward influence for Q' and vice versa. The region influenced by a block of *tape* cells will be the same for Q and Q'. The blocks of cells influenced by a block of *input* cells will differ slightly, because the direct influence will be directed at a different part of the tape. However, this does not affect the proof.

In the following, a *cycle* $\sigma(t)$ is a half-open interval (of time) $[t, \hat{t})$ such that $h_r(\hat{t}) = h_w(t)$ if $\hat{t} > t$ or such that $h_r(t) = h_w(\hat{t})$ if $\hat{t} < t$ (backward cycle). Given a time τ_1, we will be interested in nonoverlapping contiguous cycles $\sigma_1(\tau_1), \sigma_2(\tau_2), \cdots$ starting at time τ_1, such that $\sigma_1(\tau_1) = [\tau_1, \tau_2)$, $\sigma_2(\tau_2) = [\tau_2, \tau_3)$, and so on. In what follows, whenever we count cycles, the start time τ_1 either will be specified or will be clear from context and we will count the successive nonoverlapping contiguous cycles, as induced by the computation of Q. Backward cycles could alternatively be defined by using backward computations. Notice that the blocks of cells influenced by a block of input cells form a sequence of blocks, one block for each cycle.

CLAIM 3.10. *For any t, if $\hat{t} > t$ is fewer than s cycles away from t, then each cell in $Queue(\hat{t})$ is influenced by at most s input cells in $\tilde{x}\#\tilde{x}^R$.*

Proof. Let the chain of cycles starting from $\tau_1 = t$ be $\sigma_1(\tau_1)$, $\sigma_2(\tau_2), \cdots$. The proof is by induction on the indices s. No cell in $Queue(\tau_1)$ is influenced by any input cell in $\tilde{x}\#\tilde{x}^R$. During σ_1, each cell written is influenced by exactly one input cell. Suppose the claim is true for cycles σ_1 through σ_{s-1}. During the cycle $\sigma_s(\tau_s)$, each cell written is influenced by one new input cell (possibly) and by each input cell that influences the cell scanned by h_r. This adds up to at most s input cells. □

DEFINITION 3.11. For each i, we say that x_i is a *valid* block if $Queue(t'_0)$ contains a cell that is influenced by neither x_i nor $x_i{}'$.

Informally, x_i is valid if each of x_i and x_i' is read within one cycle. Indeed, if x_i is not read within one cycle, then x_i directly influences all of *Queue*(t_i) and hence influences every cell of the tape by transitivity, including every cell of *Queue*(t_0'), where t_0' is the time when h_{in} leaves x_1'.

Next, we need to show that valid blocks exist. We need the existence of only one valid block, but, in fact, the majority of blocks are valid.

CLAIM 3.12. *If there is no valid block, then Q takes* $\Omega(n^{4/3})$ *time.*

Proof. Pick a cell d on *Queue*(t_0'). Suppose there is no valid block. This means that for all i, d is influenced by either x_i or $x_i{}^R$. It means that d is influenced by at least m different cells. By Claim 3.10, we know that then the machine makes at least $m-1$ cycles from t_1 to t_0'. By Claim 3.3, the queue has length at least $n/2 - O(\log n)$ for each cycle, so the algorithm will take at least $(m-1)(n/2 - O(\log n)) \in \Omega(n^{4/3})$. □

In the following, we may assume there is at least one valid block. The next two claims explain why a valid block is a part of the input that has been coded sequentially on the queue.

CLAIM 3.13. *For each valid block* x_j*, any two cells in* x_j *influence disjoint sets of cells on the queue. Moreover, cells in* x_j' *also influence disjoint sets of cells on the queue. However, some cells on the queue can be influenced by both a cell of* x_j *and a cell of* x_j'.

Proof. If x_i is a valid block, each of x_i and x_i' must be read within one cycle. Within one cycle, each cell written into is influenced by at most one cell of x_i. This property will be preserved by transitivity throughout the successive cycles, either backward or forward. The same situation arises for x_i'. □

CLAIM 3.14. *For any time* t*, the regions influenced by the sequence of cells of a valid block* x_j *form a contiguous ordered sequence on Queue*(t). (*The same statement holds for* x_j'.)

Proof. This can be seen with a similar argument as in the previous claim. □

For our valid block x_i, both x_i and x_i' have been coded sequentially on the queue. Now we have to show that it takes $\Omega(n^{4/3}/\log n)$ time to check $x_i' = x_i{}^R$. Intuitively, we can check only a constant number of bits of x_i' at each cycle. Each cycle takes as much time as the size of the queue at that time. The strategy is to show that the size of the queue cannot decrease too much at each cycle, for each of the forward and backward computations. Then, showing that many cycles are required will provide the lower bound.

CLAIM 3.15. *If* $\hat{t} > t_{i-1}'$ *is fewer than* s *cycles away from* t_{i-1}' *and* $t < t_i$ *is fewer than* s *cycles before* t_i*, then* $|Queue(t)| + |Queue(\hat{t})| \geq n^{2/3} - O(s \log n)$.

Proof. Let x_i be a valid block, $i > 0$. Let $x_i = uv$, where u and v are strings of equal size (± 1).

If there is a time τ such that $h_{in}(\tau) \in v'$ and $h_r(\tau)$ is influenced by v, then choose $y = u$, otherwise, choose $y = v$. In both cases, for all t, if $h_{in}(t) \in y'$, $h_r(t)$ is not influenced by y. This is immediate from Claim 3.14 for the case $y = v$. For the case $y = u$, let τ be such that $h_{in}(\tau) \in v'$ and $h_r(\tau)$ is influenced by v. Let d be a cell on *Queue*(τ) not influenced by x_i or x_i'. By Claim 3.14, the region influenced by $y = u$ is after d and the region influenced by $y' = u'$ is before d (refer to Figs. 7 and 8). The regions cannot intersect.

As a consequence of our choice of y, we have that the regions influenced by y and by y' are disjoint.

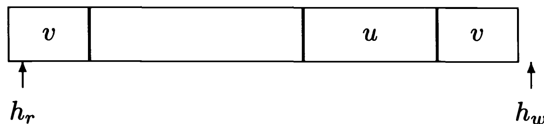


FIG. 7. *Influence of $x_i = uv$ on Queue(τ).*

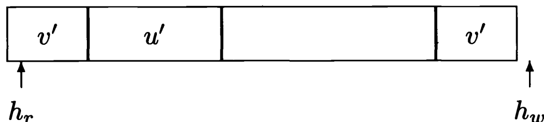


FIG. 8. *Influence of $x_i' = v'u'$ on Queue(τ).*

Let t and $\hat{t}$ be as in the statement of the claim. Let $\bar{x}$ be the string x for which y is deleted. The size of y is about $n^{2/3}$:

$$|y| \geq p/2 - 1 = \lfloor n/2m \rfloor /2 - 1 = \frac{\left\lfloor \frac{n}{2\lfloor n^{1/3}/4 \rfloor} \right\rfloor}{2} - 1 \in n^{2/3} - O(1).$$

The size of $\bar{x} = n - |y| \in n - n^{2/3} + O(1)$.

Let S be the set of cells influenced by y. We show below that x can be computed from $\bar{x}$, t, $\hat{t}$, the position of y in $\bar{x}$, the crossing sequence around S from time t to time $\hat{t}$, *Queue*(t), and *Queue*($\hat{t}$). If each item is encoded as self-delimiting, this description takes $n - n^{2/3} + O(s \log(n)) + |\mathit{Queue}(t)| + |\mathit{Queue}(\hat{t})|$ bits. Because $K(x) \geq n$, it then follows that $|\mathit{Queue}(t)| + |\mathit{Queue}(\hat{t})| \geq n^{2/3} - O(s \log(n))$.

We compute y with the information provided in the following way. For all binary strings z of equal length as y, let x_z be the string x for which z has been substituted for y. Run Q on all strings $x_z \# x_z{}^R$ until one that matches the description is found. By construction, $z = y$ matches the description. Leading to a contradiction, suppose $z \neq y$ matches the description as well. Let C_y be the accepting computation on $x \# x^R$ and C_z the accepting computation on $x_z \# x_z^R$ that matches the description. Then, by cutting and pasting the two computations we can construct a legal computation of Q on $x_z \# x^R$. Let S_z be the set of cells influenced by z in C_z. Because the crossing sequence includes the position of all heads, the regions in S and in S_z occupy the same absolute positions on the queue. Let t_y be the time when h_{in} leaves y. We can compose the accepting computation as follows. Use any of the two computations up to time t. At this time, we will have *Queue*(t).

From time t to t_y, we are dealing with backward influence. If h_w is scanning a cell of S, then h_r is scanning either a cell of S or a cell immediately before it. If h_w is scanning a cell not in S, then h_r is also scanning a cell not in S or immediately before it. Since the cell before the region has been included in the c.s., it is possible to follow C_y when h_w is in S and C_z when h_w is out of S. Notice that h_{in} cannot scan a cell of y while h_w is writing a cell out of S because of the direct influence. Moreover, $h_i n$ cannot scan a cell of z while h_w writes a cell of S because that cell would be influenced by both y and z, which cannot happen by our choice of y.

From time t_y to time $\hat{t}$,we are dealing with forward influence. Follow C_z when h_r is not in S and follow C_y when h_r is in S.

At time $\hat{t}$, the queue will correspond with the queue in both computations. Just complete the computation following any of C_y or C_z. This gives an accepting computation for a string $\notin L$, which is a contradiction. □

CLAIM 3.16. *The machine makes $\Omega(n^{4/3}/\log n)$ steps before t_i or after t'_{i-1}.*

Proof. Let T be the time Q accepts. Both $Queue(0)$ and $Queue(T)$ are of length 0. By the previous claim, $|Queue(0)| + |Queue(T)| \geq n^{2/3} - O(s \log n)$. Let $|Queue(0)| + |Queue(T)| \geq n^{2/3} - cs \log n$. This means Q makes at least $n^{2/3}/(c \log n)$ cycles for some constant c. At least $n^{2/3}/(2c \log n)$ of those cycles will have a queue of size $\Omega(n^{2/3})$, by the previous claim. This makes a total of $\Omega(n^{4/3}/\log n)$ steps. □

COROLLARY 3.17. *For off-line one-way-input one-queue machines, nondeterministic linear time is not closed under complement.*

Proof. The complement of the palindrome language used in the proof of Theorem 3.2 can be accepted in nondeterministic linear time. This can be seen as follows. If the string is of the form $w_1\#w_2$, where $|w_1| = |w_2|$, nondeterministically go and read position i of w_1 for which there is a discrepancy. While doing that, push i symbols on the queue. Then nondeterministically go and read the corresponding position of w_2. Verify the position by using the number of symbols pushed on the queue.

Other cases can be checked in deterministic linear time. Finding which case applies can be made by a nondeterministic initial move. This concludes the proof of Theorem 3.2. □

4. More queues versus fewer queues. In this section we study the power of queue machines with different numbers of queues. We first provide some straightforward upper bounds: Two queues work as well as k queues in the nondeterministic case. This motivates our research focusing on small numbers of queues. One queue can simulate k queues in quadratic time, deterministically or nondeterministically. We then provide tight, or almost tight, lower bounds for our simulations mentioned above.

4.1. Upper bounds.

THEOREM 4.1. *Two stacks can simulate one queue in linear time, for both deterministic and nondeterministic machines.*

Proof. We design a machine P with two stacks pd1, pd2. To simulate a queue, every time a symbol is pushed into the queue, P pushes the same symbol into pd1. If a symbol is taken from the queue, then P pops a symbol from pd2 if pd2 is not empty. If pd2 is empty, then P first unloads the entire contents of pd1 into pd2 and then pops the top symbol from pd2. At the end of the input, P accepts if and only if the one-queue machine accepts. □

THEOREM 4.2. *Two queues can nondeterministically simulate k queues for any fixed k in linear time.*

Proof. This theorem follows from the method used by Book and Greibach [BG70] to nondeterministically simulate k tapes by two tapes in linear time. For the sake of completeness, we will describe the idea. The two-queue machine guesses the computation of the k-queue machine and puts this guess on one queue in the form $\text{ID}_1, \text{ID}_2, \cdots$, where ID_i contains the state of the k queue machine and the $k+1$ queue symbols scanned by the k queue heads and the input head at step i. First, check that the state in each ID is consistent with the previous ID and check the correctness of the guessed input symbol in each ID_i by scanning the ID's and moving the input head when necessary. Then, scan the ID's again k times, each time simulating one of the k queues of the simulated machine on the other queue. This simulation takes $O((k+1)n) = O(n)$ time. □

THEOREM 4.3. *Three stacks can nondeterministically simulate k queues in linear time.*

Proof. Combine the ideas from the above two theorems; i.e., guess the computation of the k-queue machine as before, and put the guess into one stack. Save this guess also to another stack (but put a marker on the top). Then simulate a queue and check the correctness of the guess. (The simulation needs two stacks; one of the stacks has the guessed computation saved in the bottom.) After simulating one queue, retrieve the guessed contents; again put it into two stacks. Repeat this process for each queue. □

Remark. It is a folklore fact, and easily verified, that one-queue machines accept precisely the r.e. languages. In contrast, one-stack machines accept only CFL's. Hence, one queue is better than one stack. However, when we have more stacks, more stacks seem to be better than queues because they are more efficient. It was proved in [HM81] that four stacks can simulate a queue in real time.

THEOREM 4.4. *One queue can simulate k queues in quadratic time, both deterministically and nondeterministically.*

Proof. This is similar to the simulation of k tapes by one tape by Hartmanis and Stearns [HS65] (see [HU79, p. 292]). □

This also relates to the interesting problem of whether two heads (on one tape) are better than two tapes (each with one single head). Vitányi [Vit84a] showed that two tapes cannot simulate a queue in real time if at least one of the tape heads is within $O(n)$ cells from the start cell at all times. We saw that two stacks can simulate a queue in linear time and four stacks can do this in real time. It would be interesting to know whether two or three stacks can do this in real time. The question of how to deterministically simulate k queues by two queues in $O(n^2)$ time, like the Hennie–Stearns simulation in the tape case [HS66], remains open.

4.2. Lower bounds. We now prove optimal lower bounds for the above simulations. Let L be the following language.

$$\begin{aligned} L = \{\, & a\,\&\, b_0^1 b_1^1 \cdots b_k^1 \# b_0^2 b_0^3 b_1^2 b_2^2 b_1^3 b_3^2 \cdots b_{2i}^2 b_i^3 b_{2i+1}^2 \cdots b_{k-1}^2 b_{(k-1)/2}^3 b_k^2 \\ & b_0^4 b_{(k+1)/2}^3 b_1^4 b_2^4 b_{(k+3)/2}^3 b_3^4 \cdots b_{2i \bmod (k+1)}^4 b_i^3 b_{(2i+1) \bmod (k+1)}^4 \cdots b_{k-1}^4 b_k^3 b_k^4 \,\&\, a : \\ & b_i^1 = b_i^2 = b_i^3 = b_i^4 \text{ for } i = 0, \cdots, k \\ & \quad \text{all } b_i^j \text{ have format } \$x\$, \text{ where } x \in \{0,1\}^* \\ & \quad k \text{ is odd, and } a \in \{0,1\}^* \,\}. \end{aligned}$$

When we prove the lower bound, all the b_i^j will have the same length. The string between the first & and second & can be obtained by copying $b_0 b_1 \cdots b_k$ three times:

$$b_0 b_1 \cdots b_k \ \# b_0 b_1 \cdots b_k \ b_0 b_1 \cdots b_k,$$

and then adding one more copy of $b_0 b_1 \cdots b_k$ by inserting block b_i after $2i$ blocks, starting from $\# b_0$ in above. The superscripts on the b_i's are used only to facilitate later discussions. L can be considered as a modified version of a language used in [Maa85]. We have added a string a on both ends. The purpose of a is to prevent the queue from shrinking, since if we choose a to be a long K-random string, then before the second a is read the size of the queue has to be at least about $|a|$. We have to prevent the queue from shrinking because otherwise the crossing sequence argument would not work. In addition to the techniques in [Maa85], and [LV88], we will need the techniques introduced in this paper to treat queues.

An alternative way to describe the language L is as follows. Let y and z be sequences of blocks in which each block is of form $\$u\$$, where $u \in \{0,1\}^*$. Define $intermingle(y) =$

z if (1) the blocks of z in positions $i \equiv 2 \pmod 3$ form the string y ($z_2z_5z_8\cdots = y_1y_2y_3\cdots$) and (2) the remaining blocks of z form the string yy.

Then, $L = \{a\&y\#intermingle(y)\&a : y$ contains an even number of blocks$\}$.

THEOREM 4.5. *Simulating a deterministic two-queue machine with a one-way input tape by a nondeterministic one-queue machine with a one-way input tape requires* $\Omega(n^2/\log^2 n \log\log n)$ *time.*

Proof. We will show that the L just defined requires $\Omega(n^2/\log^2 n \log\log n)$ time on a nondeterministic one-queue machine. Since L can be trivially accepted by a deterministic two-queue machine in linear time, the theorem will follow.

Now, aiming at a contradiction, assume that a one-queue machine M accepts L in time $T(n)$, which is not in $\Omega(n^2/\log^2 n \log\log n)$. Without loss of generality, we assume that M has a binary queue alphabet and that M accepts with a final state and an empty queue. We use the same notation and definitions as in the previous section, e.g., $Queue$, $|Queue(t)|$, h_{in}, h_r, h_w, cycles, and crossing sequence.

Choose a large n and a large enough C such that $C >> |M| + c$ and all the subsequent formulas make sense, where $|M|$ is the number of bits needed to describe M and c is a constant given in Claim 4.9, which follows. Choose an incompressible string $X \in \{0,1\}^{2n}$, $K(X) \geq |X|$. Let $X = X'X''$, where $|X'| = |X''| = n$. Divide X'' into $k+1 = n/(C\log\log n)$ equal parts, $X'' = x_0x_1\cdots x_k$, where each x_i is $C\log\log n$ long. Consider a word $w \in L$, where $a = X'$, $b_i^j = x_i$ for $1 \leq j \leq 4$, and $0 \leq i \leq k$. Fix a shortest accepting path P of M on w. We will show that M takes $\Omega(n^2/\log^2 n \log\log n)$ time on P. Since n is linearly related to the size of the input, this will provide the lower bound in the theorem.[3]

Consider only the path P. Let $g(n) = C^5 \log^2 n \log\log n$. Let $t_\&$ be the time when h_{in} reaches the first &, $t'_\&$ be the time h_{in} reaches the second &, and $t_\#$ be the time when h_{in} reaches #.

CLAIM 4.6. $|Queue(t)| \geq n - O(\log n)$ *for every* $t_\& \leq t \leq t'_\&$.

Proof. The proof of this claim is the same as that of Claim 3.3 and is omitted. □

CLAIM 4.7. *The number of cycles from time* $t_\&$ *to* $t'_\&$ *is less than* $n/g(n)$.

Proof. This follows directly from the previous claim. Each cycle is of length $\Omega(n)$ and hence takes $\Omega(n)$ time. If M requires at least $n/g(n)$ cycles from $t_\&$ to $t'_\&$, then M used $\Omega(n^2/\log^2 n \log\log n)$ time, which is a contradiction. □

For each time t, we say that a substring s of the input w is *mapped into* a set S of cells on $Queue(t)$ if all the cells influenced by s on $Queue(t)$ are in S.

CLAIM 4.8. *Let* $k' = k/2 - n/g(n)$. *At time* $t_\#$, $Queue(t_\#)$ *can be partitioned into two segments,* $S_1(t_\#)$ *and* $S_2(t_\#)$, *such that* k' b_i^1*'s, say* $b_{i_1}^1, \cdots, b_{i_{k'}}^1$, *are mapped into* $S_1(t_\#)$ *and* k' *other* b_i^1*'s, say* $b_{j_1}^1, \cdots, b_{j_{k'}}^1$, *are mapped into* $S_2(t_\#)$.

Proof. Consider any cell c_0 on the $Queue(t_\#)$. By the nature of the queue and Claim 4.7, at most $m = n/g(n)$ b_i^1's can influence c_0 at $t_\#$ because M made no more than m cycles on the queue from $t_\&$ to $t_\#$. Hence, for any partition of $Queue(t_\#)$ into two parts, $S_1(t_\#)$ and $S_2(t_\#)$, there can be at most $2m$ b_i^1 blocks, each influencing both $S_1(t_\#)$ and $S_2(t_\#)$. Each of the rest of the $k+1-2m$ b_i^1 blocks either influences only $S_1(t_\#)$ or influences only $S_2(t_\#)$. It is now trivial to build S_1 and S_2 by moving the border cell by cell until the claim is satisfied. □

Now, let $S_1(t_\#)$ and $S_2(t_\#)$ be as specified in the previous claim. At any time t, let

[3]Here, as in the previous section, the language does not have a string of each length. The proof provides an input that causes the machine to take a long time for each length that has at least one string in the language. To produce a hard string for each length, just add a finite padding in the definition of the language; for example, allow markers to repeat up to four or five times.

$S_1(t)$ be the part of $Queue(t)$ influenced by $S_1(t_{\#})$ and let $S_2(t)$ be the complementary region on $Queue(t)$. Let S_1 be the set of all cells on the tape influenced by $S_1(t_{\#})$ and S_2 be the other cells.

The next claim is a simple generalization of a theorem proved in [Maa85, Thm. 3.1]. The proof of the claim is a simple reworking of the Maass proof and is hence omitted.

CLAIM 4.9. *Let S be a sequence of numbers from $0, \cdots, k$, where $k = 2^l$ for some l. Assume that every number $b \in \{0, \cdots, k\}$ is somewhere in S adjacent to the numbers $2b \pmod{k+1}$ and $2b \pmod{k+1} + 1$. Then, for every partition of $\{0, \cdots, k\}$ into two sets G and R such that $|G|, |R| > k/4$, there are at least $k/(c \log k)$ (for some fixed c) elements of G that occur somewhere in S adjacent to a number from R.* □

A $k/\sqrt{\log k}$ upper bound corresponding to the lower bound in this claim is contained in [Li88]. A more general, but weaker, upper bound can be found in [Kla84].

Remark 4.1. For each word $w \in L$, the sequence of the subscripts of the substrings (in the order they appear) in w between the # sign and the second & satisfies the requirements in Claim 4.9. For example, given k, such a sequence is formed by inserting i after $2i$th number, $i = 0, 1, \cdots, k$, in the following sequence:

$$0, 1, 2, \cdots, k, 0, 1, 2, \cdots, k.$$

Therefore, each number i is adjacent to $2i \pmod{k+1}$, and $2i + 1 \pmod{k+1}$. In what follows we will also say that a pair of b_i blocks are adjacent if their subscripts are adjacent in the above sequence.

CLAIM 4.10. *At time $t'_{\&}$, the b_i's between # and the second & are mapped into $Queue(t'_{\&})$ in the following way: either*

1. *a set, $\bar{S}_1$, of $k/(3c \log k)$ b_j's, which belong to $\{b^1_{j_1}, \cdots, b^1_{j_{k'}}\}$, are mapped into $S_1(t'_{\&})$; or*
2. *a set, $\bar{S}_2$, of $k/(3c \log k)$ b_i's, which belong to $\{b^1_{i_1}, \cdots, b^1_{i_{k'}}\}$, are mapped into $S_2(t'_{\&})$, where $c << C$ is the small constant in Claim 4.9.*

Proof. By Claim 4.7, from time $t_{\#}$ to $t'_{\&}$, M makes fewer than $n/g(n)$ cycles. Hence, h_w can alternate between S_1 and S_2 fewer than $2n/g(n)$ times. Each time h_w alternates between S_1 and S_2, h_w can map at most one adjacent pair of b^j_i blocks into both $S_1(t'_{\&})$ and $S_2(t'_{\&})$. All other pairs are each mapped totally into $S_1(t'_{\&})$ or totally into $S_2(t'_{\&})$. There are $\theta(k)$ such pairs in L.

Combining Claim 4.8, Claim 4.9, and Remark 4.1, we know that there are at least $k/c \log k - n/(C^5 \log^2 n \log \log n)$ pairs of b^j_i blocks such that each of these pairs contains a component belonging to $G = \{b^1_{i_1}, \cdots, b^1_{i_{k'}}\}$ and another component belonging to $R = \{b^1_{j_1}, \cdots, b^1_{j_{k'}}\}$. Most of these pairs, except $n/g(n)$ of them by the previous paragraph, are mapped either totally into $S_1(t'_{\&})$ or totally into $S_2(t'_{\&})$. Hence, either (1) or (2) must be true. □

Without loss of generality, assume that (1) of Claim 4.10 is true.

CLAIM 4.11. *Let t_{end} be the time M accepts. $|Queue(t_{end})| = 0$. Then there exists a time $t'_{\&} \leq t_1 \leq t_{end}$ such that $|Queue(t_1)| \leq n/(C^5 \log n)$ and from $t'_{\&}$ to t_1 M made fewer than $n/(C^5 \log n \log \log n)$ cycles.*

Proof. Otherwise M spends $\Omega(n^2/(\log^2 n \log \log n))$ time, a contradiction. □

CLAIM 4.12. *There also exists a time $t_0 \leq t_{\&}$ such that $|Queue(t_0)| \leq n/(C^5 \log n)$ and from t_0 to $t_{\&}$ M made fewer than $n/(C^5 \log n \log \log n)$ cycles.*

Proof. Note that by Claim 4.6 $|Queue(t_{\&})| \geq n - O(\log n)$. Thus, we can choose t_0 to be the last time step before $t_{\&}$ such that $|Queue(t_0)| \leq n/(C^5 \log n)$. Hence, if the claim is not true, M would spend $\Omega(n^2/(\log^2 n \log \log n))$ time, a contradiction. □

By Claim 4.7 the number of cycles M made from $t_{\&}$ to $t'_{\&}$ is less than $n/g(n)$. By Claims 4.11 and 4.12 M made at most $n/(C^5 \log n \log\log n)$ cycles from time $t'_{\&}$ to t_1 and from time t_0 to $t_{\&}$. Hence, the length of the crossing sequence at the boundary of S_1 and S_2 from $t_{\&}$ to t_1 is shorter than $n/C^4 \log n \log\log n$. For every j, if a $b_j^k \in \bar{S}_1$ for some k, then b_j^1 is mapped into S_2 by Claim 4.10.

Now we describe a program that reconstructs X with less than $|X|$ information. The program uses $Queue(t_0)$, $Queue(t_1)$, the crossing sequence around S_1, the string X where the b_j^k blocks have been deleted, and the relative position of those b_j^k blocks.

Consider every Y such that $|Y| = |X|$ and $Y = a\, y_0 \cdots y_k$ for some $y_0 \cdots y_k$.

1. Check if Y is the same as X at positions other than those places occupied by $b_j^k \in \bar{S}_1$.
2. If (1) is true, then construct the input w_Y the same way w was constructed except with x_i replaced by y_i for $i = 0, 1, \cdots, k$.
3. Copy the contents of $Queue(t_0)$ on the queue. Then simulate M from t_0 to t_1 such that h_r never goes into S_2. Whenever h_r reaches the border of S_2 it compares the current ID with the corresponding one in the crossing sequence. If they match, then M *jumps* over S_2 and, starting from the next ID on the other side of S_2, M continues until time t_1. At time t_1, compare the actual queue with what it is supposed to be. Accept Y if everything worked correctly.
4. This computation will accept if and only if $Y = X$. If it is not the case, we could compose an accepting computation on M for the string where the b_j^1 blocks correspond to those in Y and the other b_j blocks correspond to those in X. This can be done in a way very similar to what was done in Claim 3.15. The details are omitted here.

The information we used in this program is only the following:

1. $X - \bar{S}_1$, plus the information to describe the relative locations of $b_j^k \in \bar{S}_1$ in X. This would require at most

$$\begin{aligned} |X| - |\bar{S}_1||b_j^k| + O(|\bar{S}_1|\log(k/|\bar{S}_1|)) &\le 2n - |\bar{S}_1| C \log\log n + O(|\bar{S}_1| \log\log n) \\ &\le 2n - (|\bar{S}_1| C \log\log n)/2 \\ &\le 2n - n/C^2 \log n, \end{aligned}$$

   where in the first line the second term is for the b_j's in $\bar{S}_1$, the third term is for the information to describe the relative positions of $b_j \in \bar{S}_1$: To represent $|\bar{S}_1|$ elements of $\{0, 1, \cdots, k\}$, sort the elements, determine the sequence of their differences, and use a self-delimiting encoding of the natural numbers to write each difference. The final encoding has approximately $O(|\bar{S}_1|\log(k/|\bar{S}_1|))$ bits (see, for example, [LV88], [Lou84], [Eli75]).
2. Description of the crossing sequence, of length less than $n/(C^4 \log n \log\log n)$, around S_2. Again by the above efficient encoding method, this requires at most $n/(C^3 \log n)$ bits. The detail of this encoding can be found in [LV88]. The idea is as follows: Each item in the c.s. is (state of M, h_{in}'s position). Trivial encoding of $n/(C^4 \log n \log\log n)$ long c.s. needs $n/(C^4 \log\log n)$ bits. However, we can use the above method and encode only the differences of h_{in}'s positions and thus use fewer than $n/(C^3 \log n)$ bits.
3. Description of the contents of S_2 at times t_0 and t_1. But, for $i = 0, 1$ $|Queue(t_i)| \le n/(C^5 \log n)$.
4. Extra $O(\log n)$ bits to describe the program discussed above.

The total is less than $2n - n/(C \log n)$. Therefore, $K(X) < |X|$, a contradiction. □

COROLLARY. Simulating two deterministic tapes by one nondeterministic queue requires $\Omega(n^2/\log^2 n \log\log n)$.

Proof. L can also be accepted by a two-tape Turing machine in linear time. □

THEOREM 4.13. *To simulate two deterministic queues by one deterministic queue requires $\Omega(n^2)$ time.*

Proof idea. Define a language L_1 as follows ($a, x_i, y_i \in \{0,1\}^*$).

$$\begin{aligned} L_1 &= \{a \,\&\, x_1\$x_2\$\cdots\$x_k\#y_1\$\cdots\$y_l\#(1^{i_1},1^{j_1})(1^{i_2},1^{j_2})\cdots(1^{i_s},1^{j_s}) \,\&\, a \mid \\ x_p &= y_q;\ \ (p = i_1+\cdots+i_t,\ q = j_1+\cdots+j_t) \text{ and } 1 \le t \le s\}. \end{aligned}$$

L_1 can be accepted by a deterministic two-queue machine in linear time. Using the techniques in the above theorem and in [LV88], where it is proved that one deterministic Turing machine tape requires square time for this language, it can be shown that L_1 requires $\Omega(n^2)$ for a one-queue deterministic machine. We omit the proof. □

Acknowledgment. We are grateful to the referee for his careful analysis and extensive comments on the first version of this paper. We would also like to thank Andy Klapper and Roy Rubinstein for their helpful comments.

REFERENCES

[Aan74] S. O. AANDERAA, *On k-tape versus (k − 1)-tape real-time computations*, in R. M. Karp, ed., Complexity of Computation, SIAM-AMS Proceedings, Vol. 7, American Mathematical Society, Providence, RI, 1974, pp. 75–96.

[BG70] R. BOOK AND S. GREIBACH, *Quasi real-time languages*, Math. Systems Theory, 4 (1970), pp. 97–111.

[BGW70] R. BOOK, S. GREIBACH, AND B. WEGBREIT, *Time- and tape-bound Turing acceptors and afl's*, J. Comput. System Sci., 4 (1970), pp. 606–621.

[Cha77] G. CHAITIN, *Algorithmic information theory*, IBM J. Res. Develop., 21 (1977), pp. 350–359.

[DGPR84] P. DURIS, Z. GALIL, W. PAUL, AND R. REISCHUK, *Two nonlinear lower bounds for on-line computations*, Inform. and Control, 60 (1984), pp. 159–173.

[Eli75] P. ELIAS, *Universal codeword sets and representation of integers*, IEEE Trans. Inform. Theory, IT-21 (1975), pp. 194–203.

[GKS86] Z. GALIL, R. KANNAN, AND E. SZEMERÉDI, *On nontrivial separators for k-page graphs and simulations by nondeterministic one-tape Turing machines*, in Proc. 18th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 39–49.

[HM81] R. HOOD AND R. MELVILLE, *Real-time queue operations in pure LISP*, Inform. Process. Lett., 13 (1981), pp. 50–54.

[HS65] J. HARTMANIS AND R. STEARNS, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

[HS66] F. HENNIE AND R. STEARNS, *Two tape simulation of multitape Turing machines*, J. Assoc. Comput. Mach., 13 (1966), pp. 533–546.

[HU79] J. HOPCROFT AND J. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[Kla84] M. KLAWE, *Limitations on explicit construction of expanding graphs*, SIAM J. Comput., 13 (1984), pp. 156–166.

[Kol65] A. KOLMOGOROV, *Three approaches for defining the concept of information quantity*, Problems Inform. Transmission, 1 (1965), pp. 1–7.

[Kos79] S. KOSARAJU, *Real time simulation of concatenable double-ended queues by double-ended queues*, in Proc. 11th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1979, pp. 346–351.

[Li85a] M. LI, *Lower bounds by Kolmogorov complexity*, in 12th ICALP, Lecture Notes in Computer Science, No. 194, Marcel Dekker, New York, 1985, pp. 383–393.

[Li85b] ———, *Lower bounds in computational complexity*, Ph.D. thesis, Tech. Report TR85-663, Computer Science Department, Cornell University, 1985.

[Li88] ———, *Simulating two pushdowns by one tape in* $O(n^{1.5}(\log n)^{0.5})$ *time*, in Proc. 26th IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1985, pp. 56–64.

[Lou84] M. C. LOUI, *The complexity of sorting on distributed systems*, Inform. and Control, 60 (1984), pp. 70–85.

[LS81] B. L. LEONG AND J. I. SEIFERAS, *New real-time simulations of multihead tape units*, J. Assoc. Comput. Mach., 28 (1981), pp. 166–180.

[LV88] M. LI AND P. M. B. VITÁNYI, *Tape versus queue and stacks: the lower bounds*, Inform. and Comput., 78 (1988), pp. 56–85.

[LLV86] L. LONGPRÉ, M. LI, AND P. M. B. VITÁNYI, *The power of the queue*, in Structure in Complexity Theory, Lecture Notes in Computer Science, Springer-Verlag, 223, 1986, pp. 219–233.

[Maa85] W. MAASS, *Combinatorial lower bound arguments for deterministic and nondeterministic Turing machines*, Trans. Amer. Math. Soc., 292 (1985), pp. 675–693.

[MSS87] W. MAASS, G. SCHNITGER, AND E. SZEMEREDI, *Two tapes are better than one for off-line Turing machines*, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 94–100.

[Pau82] W. PAUL, *On-line simulation of* $k+1$ *tapes by* k *tapes requires nonlinear time*, Inform. and Control, 53 (1982), pp. 1–8.

[PSS81] W. PAUL, J. SEIFERAS, AND J. SIMON, *An information-theoretic approach to time bounds for on-line computation*, J. Comput. System Sci., 23 (1981), pp. 108–126.

[Sol64] R. SOLOMONOFF, *A formal theory of inductive inference*, Part 1 and Part 2, Inform. and Control, 7 (1964), pp. 1–22, 224–254.

[Vit84a] P. M. B. VITÁNYI, *On two-tape real-time computation and queues*, J. Comput. System Sci., 29 (1984), pp. 1303–1311.

[Vit84b] ———, *One queue or two pushdown stores take square time on a one-head tape unit*, Tech. Report CS-R8406, Computer Science, CWI, Amsterdam, 1984.

[Vit85] ———, *An N**1.618 lower bound on the time to simulate one queue or two pushdown stores by one tape*, Inform. Process. Lett., 21 (1985), pp. 147–152.

SIAM J. COMPUT.
Vol. 21, No. 4, pp. 713–732, August 1992

© 1992 Society for Industrial and Applied Mathematics
006

MAXIMUM SIZE OF A DYNAMIC DATA STRUCTURE: HASHING WITH LAZY DELETION REVISITED*

DAVID ALDOUS[†], MICHA HOFRI[‡], AND WOJCIECH SZPANKOWSKI[§]

Abstract. The dynamic data structure management technique called *hashing with lazy deletion* (HwLD) is studied. A table managed under HwLD is built by a sequence of insertions and deletions of items. When hashing with lazy deletions, one does not delete items as soon as possible but keeps more items in the data structure than would be the case with *immediate-deletion* strategies. This deferral allows the use of a simpler deletion algorithm, leading to a lower overhead—in space and time—for the HwLD implementation. It is of interest to know how much extra space is used by HwLD. This paper investigates the maximum size and the excess space used by HwLD, under general probabilistic assumptions, by using the methodology of queueing theory. In particular, for the Poisson arrivals and general lifetime distribution of items, the excess space does not exceed the number of buckets in HwLD. As a byproduct of the analysis, the limiting distribution of the maximum queue length in an $M|G|\infty$ queueing system is also derived. The results generalize previous work in this area.

Key words. dynamic dictionary storage, hashing with lazy deletion, maximum queue length, $M|G|\infty$ queue

AMS(MOS) subject classifications. 60K30, 68A50

1. Introduction. The purpose of this paper is to present a thorough analysis of *hashing with lazy deletion* (HwLD) in a general probabilistic framework. An item arrives at a hashing table and needs to be stored for some period (the item's *lifetime*). Different probability models for arrival and lifetimes are discussed later. We always assume that the assignment of items to the H buckets of the hashing table is uniform: That is, each item has probability $1/H$ to select each bucket, independent for different items and independent of the arrival and lifetimes.

The strategy of HwLD was proposed by Van Wyk and Vitter [22]. The principle of HwLD is very simple: An item in a bucket is not deleted as soon as possible (i.e., when its lifetime expires). Instead, the item is removed at the first arrival to the item's bucket after the item's expiration time. The point is that algorithms that delete items as soon as possible may have unacceptably high overhead, even though they require less storage space for the items themselves. In other words, there is a tradeoff between the time overhead incurred by immediate deletions and the space overhead that accrues if the time overhead is kept small. For more details concerning HwLD and its applications the reader is referred to [16]–[18], [22].

A natural problem is to examine how much storage space HwLD requires and to compare it with the storage space of a standard hashing strategy that we shall call *hashing with immediate deletion* (HwID). A particularly intriguing problem is to estimate the amount of excess space used by HwLD. Let $U_H(t)$ and $N_H(t)$ denote the number of items at time t in a table with H buckets, used for HwLD and HwID, respectively; think of this notation as a mnemonic for the "used" and "needed" amounts of space. The term "table size" will be conventionally used to denote either of these quantities.

* Received by the editors July 25, 1990; accepted for publication (in revised form) July 19, 1991.

[†] Department of Statistics, University of California, Berkeley, California 94720. The research of this author was supported by National Science Foundation grant MCS87-01426.

[‡] Department of Computer Science, The Technion, 32000 Haifa, Israel. Present address, Department of Computer Science, University of Houston, Houston, Texas 77204.

[§] Department of Computer Science, Purdue University, West Lafayette, Indiana 47907. The research of this author was supported by Air Force Office of Scientific Research grant 90-0107, in part by National Science Foundation grant CCR-8900305, and in part by grant R01 LM05118 from the National Library of Medicine.

Let $W_H(t) \equiv U_H(t) - N_H(t)$ be the space that the HwLD wastes at time t. We investigate the (expected) instantaneous difference $E[W_H(t)]$ and the difference between $E \max_{0 \le t \le T} U_H(t)$ and $E \max_{0 \le t \le T} N_H(t)$. These two differences are called the (expected) "wasted space" and "excess space," respectively. Also, there is interest in evaluating $\max_{0 \le t \le T} N_H(t)$ and $\max_{0 \le t \le T} U_H(t)$ themselves. To motivate this further we note—after Van Wyk and Vitter [22]—that $N_H(t)$ can be interpreted as the number of "live" items at time t, regardless of the hashing strategy implementation. In other words, $N_H(t)$ is the minimum space requirement for any algorithm that maintains $N_H(t)$ items in the data structure at time t. For such problems the quantity $\max_{0 \le t \le T} N_H(t)$ is a lower bound on the space requirement, and $\max_{0 \le t \le T} U_H(t)$ is the corresponding space used by hashing with lazy deletion. We shall show that both display similar growth with respect to the traffic intensity and time. Furthermore, the difference $\max_{0 \le t \le T} U_H(t) - \max_{0 \le t \le T} N_H(t)$ will be shown to be small in a sense we detail later. Hence, the HwLD strategy can be said to be *near optimal* in terms of storage-space requirements [22] and very attractive in terms of time complexity because of its low overhead cost. We study these and some related questions in this paper (cf. also [2]).

Although this paper adopts a queueing-theoretical approach, its approach differs from the traditional queueing analyses in some important aspects. Our look at the problem resembles that by Morrison, Shepp, and Van Wyk [16]; that is, we first consider a model suitable for a single bucket, and then we analyze the complete model, involving a (finite) number of such buckets. We use a natural sample-path approach that readily gives answers concerning the average wasted space problem in HwLD. To study the *excess* space we have to evaluate the maximum queue length in $GI|G|\infty$ queueing systems,[1] and we prove some new results concerning this maximum. In passing, we note that although we consider only hashing tables, the evaluation of maximum queue lengths might be useful for the analysis of several other data structures. Our methodology can be applied in studying dynamics of data structures that share some features with *queues*, namely, structures that are built during a sequence of insertions and deletions [9], [14], [15]. We mention here dictionaries, linear lists, stacks, priority queues, and symbol tables [3].

The literature on HwLD is rather scanty. As mentioned previously, HwLD was introduced by Van Wyk and Vitter [22]. Under exponential/exponential interarrival/lifetimes assumptions ($M|M|\infty$ model) they proved that $EU_H(t) - EN_H(t) = H$. For the same model, Morrison, Shepp, and Van Wyk [16] estimated *numerically* the distribution of $\max_{0 \le t \le T} U_H(t)$, and from these numerical analyses they conjectured that the difference $E\{\max_{0 \le t \le T} U_H(t)\} - E\{\max_{0 \le t \le T} N_H(t)\} = O(H)$. In two recent papers Mathieu and Vitter [17], [18] proved this conjecture for an $M|G|\infty$ model by using an interesting (and rather complicated) probabilistic approach. In addition, [18] establishes the rate of growth for the maximum queue length in an $M|G|\infty$ model. Some preliminary results concerning HwLD are also presented in Szpankowski [21]. Our results provide generalizations in various directions. First, we investigate the most general $GI|G|\infty$ model and obtain basic results in this setting. In particular, we show how they differ from the $M|G|\infty$ model. We prove—as conjectured—that indeed $EU_H(t) - EN_H(t) = H$ in the $M|G|\infty$ model of HwLD (see also [18]) but *not* in the $GI|M|\infty$ model (Theorem 1). Next we consider the maximum table size under HwLD and prove that in general

[1] A typical single queueing model is that of $GI|G|c$, where the first G stands for general (arbitrary) interarrival time distribution of items (customers), the second G denotes the general (arbitrary) lifetime distribution, and the final c represents the number of servers. When an I is affixed to the first G it signifies that the interarrival duration distribution is sampled independently each time. Finally, $M|G|\infty$ denotes the specialization in which the arrival time process is Poisson with rate λ, and $GI|M|\infty$ denotes the specialization in which the lifetime distribution is exponential (μ), with an infinite number of servers [13].

$\max_{1\le k\le n} U_H(\tau_k) = o(\log n)$, where τ_k is the arrival time for the kth item, and in particular for $M|G|\infty$ (see also [18]) $\max_{1\le k\le n} U_H(\tau_k) \sim \log n/\log\log n$ (Theorem 2). Finally, we deal with the excess space and prove that in the $M|G|\infty$ model of HwLD, $\Pr\{\max_{1\le k\le n} U_H(\tau_k) - \max_{1\le k\le n} N_H(\tau_k) > H+2\} \to 0$ as $n\to\infty$ [Theorem 3(i)]. To derive this result we need to obtain sharp asymptotics for the distribution of the maximum queue length in an $M|G|\infty$ queue (Theorem 6), which seems to be a new result. We have also one result on the excess space for the general model without any probability assumptions on arrival and lifetimes. For large H and n *polynomially* large in H, we show that $\Pr\{\max_{0\le k\le n} U_H(\tau_k) - \max_{0\le k\le n} N_H(\tau_k) \ge H + O(\sqrt{H\log n})\} = o(1)$ for large n [Theorem 3(ii)].

The paper is organized as follows. In the next section we formulate a probabilistic model of HwLD and state our results. Section 3 contains the proofs of those results that deal with the maximum size. These proofs require us to investigate the asymptotic distribution of the maximum queue length in a queueing system with an infinite number of servers. Finally, in §4 we sketch future research directions that aim to get a more realistic approach to the maximum size of dynamic data structures.

2. Statement of results. We consider a table managed under HwLD with H buckets. Items arrive at arbitrary times $0 \le \tau_1 < \tau_2 < \cdots$. Let $\eta(t)$ represent the number of arrivals up to time t. An arriving item selects one out of the H buckets at random (with uniform probability) and joins the items assigned to this bucket. The kth item has a lifetime (required storage time) $S_k > 0$. Under HwID, the kth item is removed at time $\tau_k + S_k$. Let $N_H(t)$ be the total number of items in the hash table at time t under HwID, and let $N_H^{(i)}(t)$ be the number in bucket i. Under the HwLD scheme with the same arrival and lifetimes, let $U_H(t)$ be the total number of items in the hash table at time t and let $U_H^{(i)}(t)$ be the number in bucket i. Let $W_H(t) = U_H(t) - N_H(t)$ denote the wasted space.

From the verbal description of HwLD and Fig. 1, we see the following *sample path* relationship in each bucket i and at each time t:

$$U_H^{(i)}(t) = N_H^{(i)}\left(\tau_{\eta(t)}^{(i)}-\right) + 1, \tag{2.1}$$

where $\tau_{\eta(t)}^{(i)}$ denotes the time of the last arrival to bucket i before time t. Note that, strictly speaking, (2.1) holds only after the first arrival of a customer to the queue. Thus, $N_H^{(i)}\left(\tau_{\eta(t)}^{(i)}-\right)$ denotes the number of items in bucket i with unexpired lifetimes immediately before the time of the last arrival to bucket i before t (i.e., the number *seen* by that arrival). Summing over buckets,

$$U_H(t) = \sum_{i=1}^{H} N_H^{(i)}\left(\tau_{\eta(t)}^{(i)}-\right) + H. \tag{2.2}$$

Equations (2.1) and (2.2) are similarly restricted: (2.2) holds only after every bucket has had an arrival. Thus, (2.2) expresses the number $U_H(t)$ of items used by HwLD in terms of the queue-length processes $N_H^{(i)}(t)$ in individual buckets with immediate deletion.

So far we have made no assumptions about the arrival and lifetimes, and in this generality we have only one result [Theorem 3(ii)]. For the other results we introduce probabilistic models for the arrival and lifetimes. In the $GI|G|\infty$ model the interarrival times $\xi_k = \tau_k - \tau_{k-1}$ are assumed to be strictly positive independent and identically distributed (i.i.d.) random variables with mean $1/\lambda$, and the lifetimes S_k are also assumed to be strictly positive i.i.d. with mean $1/\mu$. Let $\rho = \lambda/\mu$ denote the traffic intensity.

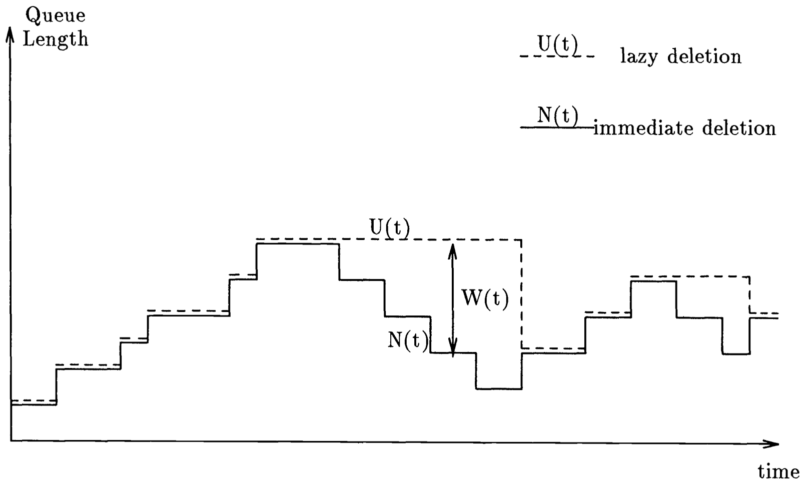


FIG. 1. *Relationship between $N_H(t)$ and $U_H(t)$ in a single bucket.*

We state our results for the *stationary* versions of these processes. An alternative is to assume the table starts empty. The results about asymptotic maxima [Theorem 2 and Theorem 3(i)] are unchanged, whereas Theorem 1 would hold with N_H and U_H interpreted as the limit ($t \to \infty$) in distribution of $N_H(t)$ and $U_H(t)$. Note that this limit exists under weak technical assumptions using regeneration arguments [11].

It is important to note that the processes $N^{(i)}(t)$ are in general *dependent* as the bucket i varies (and similarly for $U^{(i)}(t)$). The $M|G|\infty$ model is an exception: by the "independent sampling" property of the Poisson arrival process, what happens in different buckets is *independent*.

Now we are ready to present our results concerning HwLD. We concentrate on comparing it with HwID.

THEOREM 1 (STATIONARY DISTRIBUTION AND MOMENTS OF THE TABLE CONTENT). *Consider the stationary $GI|G|\infty$ model of HwLD. Let U_H and N_H be the limiting random variables for $U_H(t)$ and $N_H(t)$. Let $N^{(i)}(\tau^{(i)}-)$ denote the number of items seen in bucket i by an item arriving in bucket i in the corresponding immediate deletion model.*

(i) *In the $GI|G|\infty$ model,*

$$\Pr\{U_H = k + H\} = \Pr\left\{\sum_{i=1}^{H} N^{(i)}(\tau^{(i)}-) = k\right\}, \qquad k \geq 0 \tag{2.3a}$$

and

$$EU_H = H(1 + EN^{(i)}(\tau^{(i)}-)) \quad \textit{for any fixed } i. \tag{2.3b}$$

(ii) *In the $M|G|\infty$ model, $U_H - H$ and N_H each have Poisson (ρ) distribution; that is,*

$$\Pr\{U_H = k + H\} = e^{-\rho}\frac{\rho^k}{k!}, \qquad k \geq 0. \tag{2.3c}$$

So, in particular,

$$EU_H = EN_H + H, \tag{2.3d}$$

(2.3e)
$$\mathrm{var} U_H = \mathrm{var} N_H.$$

(iii) *In the $GI|M|\infty$ model,*

(2.3f)
$$EU_H = \frac{1}{\rho}\frac{A^*(\mu)}{1 - A^*(\mu)}EN_H + H,$$

where $A^*(\mu) = Ee^{-\mu\xi}$ and ξ is the interarrival time.

Remark. Note that (2.3f) implies that (2.3d) does *not* in general hold for non-Poisson arrival processes.

Proof. Part (i) is immediate from (2.2). For (ii), N_H has the stationary distribution of the $M|G|\infty$ queue, which is well known to be Poisson (ρ). Applying (2.1) to bucket i and using the PASTA[2] property, we see that $U^{(i)} - 1$ has Poisson (ρ/H) distribution. Summing over buckets (and using independence between buckets) gives (2.3c), which immediately implies (2.3d, 2.3e). This was also obtained in [18] in a rather different and more complicated manner.

To prove (iii), note that in the $GI|M|\infty$ queue $EN_H = \rho$ [23, p. 348]. So, in view of (2.3b), what we need to show is

$$EN^{(i)}(\tau^{(i)}-) = \frac{A^*(\mu)/H}{1 - A^*(\mu)}.$$

Now the immediate-deletion process in the given bucket i is the $GI|M|\infty$ queue with a different interarrival time, say $\hat{\xi}$. Let $\hat{A}^*(u) = Ee^{-u\hat{\xi}}$. A standard computation [23, p. 348] (conditioning on all previous arrival times) gives the probability-generating function (pgf) of $N^{(i)}(\tau^{(i)}-)$:

$$Ez^{N^{(i)}(\tau^{(i)}-)} = \exp\left\{\frac{\hat{A}^*(\mu)}{1 - \hat{A}^*(\mu)}\log z\right\}.$$

In particular,

$$EN^{(i)}(\tau^{(i)}-) = \frac{\hat{A}^*(\mu)}{1 - \hat{A}^*(\mu)}.$$

Now, $\hat{\xi}$ is a sum of a geometrically distributed number of interarrival times ξ_j:

$$\hat{\xi} = \sum_{j=1}^{G} \xi_j\,; \qquad P(G = g) = H^{-1}(1 - 1/H)^{g-1}, \qquad g \geq 1,$$

and a brief calculation gives

$$Ee^{-\mu\hat{\xi}} = \frac{A^*(\mu)/H}{1 - A^*(\mu)(1 - 1/H)}.$$

Substituting this into the previous formula leads to the desired equation, completing the proof of Theorem 1. □

[2] PASTA stands for *Poisson Arrivals Sees Time Average*, and this implies that the time-stationary distribution of the queue length is the same as the customer-stationary distribution, that is, as seen by an arriving customer. More details can be found in [23], mainly in §5.16.

Our main results concern the maximum table size over long time intervals. Note that the time of attainment of the maximum (for either $N_H(t)$ or $U_H(t)$) must occur immediately after some arrival. Thus, we can state Theorems 2 and 3 in terms of maxima seen at arrival times, and the results remain true also if we interpret the maxima as taken over the entire corresponding time intervals—up to a difference of one, since in the latter case the arrival is counted as well.

The first result of this type defines the order of growth with time of the maximum occupancy of the table using HwLD. The proof is given in §3. To review some standard notation, $a_n \sim b_n$ means $a_n/b_n \to 1$, and for random variables $X_n \to 0$ *in probability* (pr.) means $\Pr\{|X_n| > \varepsilon\} \to 0$ as $n \to \infty$ for any fixed $\varepsilon > 0$. The symbol $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x.

THEOREM 2 (MAXIMUM SIZE OF A TABLE UNDER HWLD). (i) *For an $M|G|\infty$ model of HwLD, suppose the lifetime S satisfies $ES\log^2 S < \infty$. Then,*

$$\Pr\{\lfloor a_n \rfloor + 1 \le \max_{1\le k\le n} U_H(\tau_k) \le \lfloor a_n \rfloor + 1 + H\} \to 1 \quad \text{as } n \to \infty, \tag{2.4a}$$

where $\{a_n\}$ is a particular sequence defined below, which satisfies $a_n \sim \log n/\log\log n$.

(ii) *For a $GI|G|\infty$ model of HwLD the maximum table size satisfies* $\max_{1\le k\le n} U_H(\tau_k) = o(\log n)$ *in probability; precisely, as $n \to \infty$*

$$\frac{1}{\log n}\max_{1\le k\le n} U_H(\tau_k) \to 0 \qquad (pr.), \tag{2.4b}$$

provided the lifetime S satisfies $\Pr\{S > x\} = O(e^{-\beta x})$ *for some $\beta > 0$.*

Remark. Part (i) implies that $\max_{1\le k\le n} U_H(\tau_k) \sim \log n/\log\log n$ (pr.) (also obtained in [18]). It is plausible that the conclusion $\max_{1\le k\le n} U_H(\tau_k) \sim c\log n/\log\log n$ (pr.), for some $c > 0$, also holds for the $GI|G|\infty$ model under weak assumptions on interarrival and service times.

The next finding is our strongest result, and it estimates the *excess space* that HwLD requires in order to accommodate the same arrival process as HwID. This result resolves some open problems posed in [22] and [16]. It also says that under fairly general assumptions HwLD is *near optimal.* Indeed, we prove the following.

THEOREM 3 (LIMITING EXCESS SPACE). (i) *In the stationary $M|G|\infty$ model of HwLD, as $n \to \infty$*

$$\Pr\{\max_{1\le k\le n} U_H(\tau_k) - \max_{1\le k\le n} N_H(\tau_k) > H + 2\} \to 0, \tag{2.5a}$$

provided the lifetime S satisfies $ES\log^2 S < \infty$.

(ii) *Consider* HwLD *with arbitrary (i.e., no probabilistic assumptions) arrival and lifetimes. Then for $n, H \ge 2$, and $b > H$,*

$$\Pr\{\max_{1\le k\le n} U_H(\tau_k) - \max_{1\le k\le n} N_H(\tau_k) \ge b\} \le 2n\left(\frac{H+b}{2b}\right)^{b/2}\left(\frac{H+b}{2H}\right)^{H/2}. \tag{2.5b}$$

If H is large and n is at most polynomially large in H, then the bound on the difference is $H + O(\sqrt{H}\log n)$. In particular, $\Pr\{\max_{0\le k\le n} U_H(\tau_k) - \max_{0\le k\le n} N_H(\tau_k) > H + (2+\varepsilon)\sqrt{H\log n}\} = o(1)$, *for any $\varepsilon > 0$.* □

In summary, our results indicate that HwLD should provide a very attractive alternative solution for hashing implementations. In particular, under fairly general probabilistic assumptions, the average storage space required by HwLD is *not* much larger

than for ordinary HwID (Theorem 1). We would assume this observation to hold for a wider range of probabilistic models than those for which we could construct a proof. Furthermore, with very high probability, the excess space incurred by lazy deletion is relatively small compared with the space requirements of HwID (Theorem 3). Although it increases with the lifetime of the system, the rate of growth $O(\sqrt{\log n})$ is reassuringly moderate. Since HwLD allows us to use data structures that have low space overhead, we are led to the conclusion that HwLD is *essentially optimal* in terms of space and time complexity. Note, however, that with small probability something may still go wrong with HwLD. Indeed, it is not difficult to create realizations in which the arrival and lifetime processes interact to have time points at which the wasted space, i.e., the difference $U_H(t) - N_H(t)$ assumes arbitrarily large values.

Finally, one usually interprets $n \to \infty$ asymptotics as approximations for large finite n. The results we report here sometimes need a more precise statement about the relation between the parameters. For example, some results would require n to be "superexponentially large in ρ" for the approximation to be valid. In such a case the asymptotic results have limited practical importance. We shall comment on this difficulty and suggest an alternative approach for its resolution in our concluding remarks in §4.

3. Analysis of the maximum size. In this section we prove Theorems 2 and 3 stated above. Both theorems deal with the maximum size of a table under HwLD. In the course of deriving these results we present some new findings concerning an asymptotic distribution of the maximum queue length in an $M|G|\infty$ queue (Theorem 6).

3.1. Maximum size of HwLD. To obtain the required bounds on the table size under HwLD, the following lemma, corollary, and theorem show progressively tighter bounds on the maxima of sequences of identically distributed random variables. Lemma 4 and its Corollary 5 are a direct consequence of Anderson's findings [5], but we bring them here for convenience of reference.

LEMMA 4. *Let $X_1, X_2, \cdots$ be identically distributed* discrete, possibly dependent *random variables with common marginal distribution function* $F(x) = \Pr\{X < x\}$, *where x belongs to the set $\mathcal{N}$ of nonnegative integers. We denote* $M_n \equiv \max_{1\le k\le n} X_k$.

(i) *Let*

$$F(x) < 1 \quad \text{for } x < \infty, \tag{3.1}$$

and assume a function $g(x,b)$ exists, such that for any positive integer $b \in \mathcal{N}^+$ the distribution function $F(x)$ satisfies

$$\frac{1 - F(b+x)}{1 - F(x)} = g(x,b), \tag{3.2}$$

where $\lim_{x\to\infty} g(x,b) = 0$ (*that is, the distribution of X_i has a superexponential tail*). *Also, let a_n be the smallest solution of the following characteristic equation*[3]

$$n[1 - F(a_n)] = 1. \tag{3.3}$$

Then,

$$\Pr\{M_n \ge \lfloor a_n \rfloor + 1 + b\} = O(g(a_n, b)) \to 0, \qquad n \to \infty. \tag{3.4}$$

[3] Since the distribution function is only piecewise continuous with jumps at the integers, (3.3) may not be satisfiable for any n. We define then a "solution" of (3.3) by embedding the discrete random variables in a continuous version with a distribution that coincides with $F(x)$ at the integers. Following [5], let $G(x) = 1 - F(x)$, $h(n) = -\log G(n)$, and $h_c(x) \equiv h(\lfloor x \rfloor) + (x - \lfloor x \rfloor)(h(\lfloor x \rfloor + 1) - h(\lfloor x \rfloor))$. Then the continuous complementary distribution $G_c(x) \equiv \exp[-h_c(x)]$ is the function we use; a_n is the solution of $G_c(a_n) = 1/n$.

In other words, $M_n \leq \lfloor a_n \rfloor + 1$ *(pr.).*

(ii) *If* $X_1, X_2, \cdots, X_n$ *are independent random variables satisfying the above hypotheses, then*

$$\Pr\{M_n < x\} - \exp(-n[1 - F(x)]) \to 0 \quad \textit{as } n, x \to \infty \tag{3.5a}$$

and

$$\Pr\{M_n = \lfloor a_n \rfloor + 1 \textit{ or } \lfloor a_n \rfloor\} = 1 - O(g(a_n, 1)) \to 1 \quad \textit{as } n \to \infty, \tag{3.5b}$$

where a_n *solves* (3.3).

Proof.

(i) Equation (3.4) follows directly from Boole's inequality and the superexponentiality assumption (3.2); namely, for $b \in \mathcal{N}^+$

$$\Pr\{M_n \geq \lfloor a_n \rfloor + b + 1\} \leq n \cdot [1 - F(\lfloor a_n \rfloor + b + 1)] = O(g(a_n, b)) \to 0$$

when $a_n \to \infty$, which follows from (3.1) and (3.2). This sequence $\{a_n\}$ is the one used in the formulation of Theorem 2.

(ii) Let $G(x) = 1 - F(x)$. Equation (3.5a) follows immediately from the observation that $\Pr\{M_n < x\} = F^n(x)$ and developing it as

$$\Pr\{M_n < x\} - e^{-nG(x)} = e^{-nG(x)}(e^{-nG^2(x)(1/2+G(x)/3+\ldots)} - 1).$$

It can be seen that either of the two factors on the right-hand side vanishes as x or n increases.

For (3.5b) we note that since M_n assumes integer values only, we may write

$$\Pr\{M_n < \lfloor a_n \rfloor\} = \Pr\{M_n < \lfloor a_n \rfloor - \varepsilon\} \leq \Pr\{M_n < a_n - \varepsilon\}$$

for some $0 < \varepsilon < 1$ whether a_n is integer or not. Then from relation (3.5a) we have for n large enough, where $G_c(x)$ is a continuous version of $G(x)$ (see footnote 3),

$$\Pr\{M_n < \lfloor a_n \rfloor\} \leq \exp\{-nG_c(a_n - \varepsilon)\} = \exp\left\{-\frac{G_c(a_n - \varepsilon)}{G_c(a_n)}\right\}.$$

Since for $G_c(x)$ the analogue of equation (3.2) holds for any $b > 0$, the last argument in braces is unbounded as $n \to \infty$, and hence

$$\Pr\{M_n < \lfloor a_n \rfloor\} = o(1).$$

This, together with part (i), imply the result. □

As a direct consequence of the above we show the following corollary concerning the maximum of the Poisson process.

COROLLARY 5.

(i) *Let* $\{X_k; k \geq 1\}$ *be (possibly dependent) Poisson* (ρ) *variables. Let* $I(n)$ *be a random sequence possibly dependent on the* $\{X_k\}$, *with* $I(n)/n \to c$ *(pr.) as* $n \to \infty$, *for some finite* $c > 0$. *Then there exists (an increasing) sequence* x_n *satisfying the following:*

$$\Pr\{\max_{1 \leq k \leq I(n)} X_k < x_n\} \to 1.$$

(ii) *If* $\{X_k, k \geq 1\}$ *are i.i.d. Poisson* (ρ) *distributed random variables, then for large enough integers* a *and* n

$$\Pr\left\{\max_{1\leq k\leq n} X_k < a\right\} - \exp(-ne^{-\rho}\rho^a/a!) \to 0 \quad \text{as } n, a \to \infty, \tag{3.6a}$$

and

$$\Pr\left\{\max_{1\leq k\leq n} X_k = \lfloor a_n \rfloor + 1 \quad \text{or} \quad \lfloor a_n \rfloor\right\} = 1 - O(1/a_n) \to 1 \quad \text{as } n \to \infty. \tag{3.6b}$$

For large n *the sequence* $\{a_n\}$ *satisfies*

$$a_n \sim \frac{\log n - \rho}{\log(\log n - \rho) - \log \rho} \sim \frac{\log n}{\log\log n}, \tag{3.6c}$$

where a_n *is defined as the smallest solution of the equation*

$$n \cdot \frac{\gamma(a_n, \rho)}{\Gamma(a_n)} = 1. \tag{3.7a}$$

In the above, $\gamma(x, \rho) \equiv \int_0^\rho t^{x-1}e^{-t}dt$ *is the incomplete gamma function and* $\Gamma(x) = \gamma(x, \infty)$ *is the gamma function* [1].

Remark. By using the property $P(X \geq x) \sim P(X = x)$ we can specify the sequence in (3.6b) in an alternative way. Namely, (3.6b) holds with $\lfloor a_n \rfloor$ replaced by any integer-valued sequence a_n satisfying

$$ne^{-\rho}\rho^{a_n+2}/(a_n+2)! \to 0 \quad \text{and} \quad ne^{-\rho}\rho^{a_n}/(a_n)! \to \infty, \tag{3.7b}$$

and $a_n \to \infty$.

Proof. Part (i) follows from the same arguments as in the proof of Theorem 3.2 in Berman [6] (see also [5, pp. 109–111], [10, Chap. 6.2]). In particular, by using the partition arguments of [6] for any $\varepsilon > 0$, we obtain $\Pr\{\max_{1\leq k\leq I(n)} X_i > x_n\} \leq 2\varepsilon + nc(1+\varepsilon)\Pr\{X_i > x_n\}$, as in the proof of Lemma 4(i). Putting $x_n = \lfloor a_{\lfloor nc \rfloor} \rfloor + 2$, with a_n given by (3.6c), we establish part (i).

For part (ii), equation (3.6a) is immediate from (3.5a) on observing that $\Pr\{X \geq x\} \sim \Pr\{X = x\}$ as $x \to \infty$ (because of superexponentiality of the Poisson distribution). Equation (3.6b) is identical to (3.5b), and for the value of a_n one needs only to notice that the tail of the Poisson distribution can be computed as (cf. [8]).

$$1 - F(x) = \Pr\{X \geq x\} = \sum_{j=x}^{\infty} \frac{\rho^j}{j!}e^{-\rho} = \frac{\gamma(x, \rho)}{\Gamma(x)}.$$

For an asymptotic solution of (3.7a) we follow [1, p. 262] and approximate the incomplete gamma function as $\gamma(x, \rho) \approx \Gamma(x)e^{-\rho}\rho^x/\Gamma(x+1)$. Hence, for large n (3.7a) reduces to

$$n \cdot \frac{e^{-\rho}\rho^{a_n}}{\Gamma(a_n+1)} = 1. \tag{3.7c}$$

Applying Stirling's formula to the above, one finds

$$\left(\frac{a_n}{\rho e}\right)^{a_n} = \frac{e^{-\rho}}{\sqrt{2\pi}} \frac{n}{\sqrt{a_n}}.$$

This equation can be solved for large n by *asymptotic bootstrapping*, and this leads to equation (3.6c). Finally, the evaluation of the function $g(x,1)$ from Lemma 4 gives $g(x,1) = \rho x!/(x+1)! \sim \rho/x$, and this gives (3.6b). □

To prove Theorem 2(i) and Theorem 3(i) we need sharp asymptotic estimates for the maximum queue length in an $M|G|\infty$ queue. Recall that the queue length in $M|G|\infty$ has the stationary Poisson (ρ) distribution; however, the dependence of queue sizes at different times precludes the simple-minded use of Corollary 5. Note also that the queue-length process in *not* Markov. We shall prove the following theorem and show that it, together with Theorem 3(i), directly implies Theorem 2(i).

THEOREM 6. *Let X_t be the queue length in the stationary $M|G|\infty$ queue. Then, uniformly in t_0,*

$$\Pr\left\{\sup_{t\le t_0} X_t \le a\right\} - \exp(-t_0\lambda e^{-\rho}\rho^a/a!) \to 0 \quad \text{as } a\to\infty \tag{3.8a}$$

Now, let X_{τ_k-} be the queue length X_t just before the arrival time τ_k. Then, with the sequence a_n that provides the solution to (3.7a) *we find*

$$\Pr\left\{\max_{1\le k\le n} X_{\tau_k-} = \lfloor a_n\rfloor + 1 \text{ or } \lfloor a_n\rfloor\right\} \to 1 \quad \text{as } n\to\infty, \tag{3.8b}$$

provided the lifetime S satisfies $ES\log^2 S < \infty$.

Remark. One could reformulate (3.8b) to refer to a maximum on a time interval T. This would lead to a similar right-hand side with a_n replaced by $a_{\lfloor\lambda T\rfloor}$.

Proof. Consider first relation (3.8a). Fix an integer a. Call a time t with $X_{t-} = a$ and $X_t = a+1$ an *upcrossing time*. Classify items in the queue as "cleared" or "uncleared" according to the following rules: (i) Each new arrival is "uncleared." (ii) Whenever the number of "uncleared" items increases to $a+1$, all these $a+1$ items are declared "cleared" (call such a time a *clearing time*). There is a stationary version of this process, and for this stationary version define X_t = total number of items at time t and X_t^* = number of uncleared items at time t.

Of course, X_t by itself is the $M|G|\infty$ queue. Also, (X_t^*) by itself can be regarded as the process that behaves like the $M|G|\infty$ queue with the following modification: When an arrival makes the queue length equal to $a+1$, all items in storage are removed. The purpose of the joint construction is to obtain the following property: (iii) The set of clearing times for X_t^* is a subset of the set of upcrossing times for X_t. To see why, let t_1 be a clearing time and let t_0 be the last time before t_1 that the queue was empty. Then $X_t = X_t^*$ on $t_0 \le t < t_1$, so t_1 is an upcrossing time for X_t.

Write $q(a)$ for the chance that a typical upcrossing time of X is a clearing time of X^*. Then

$$q(a) = \frac{\text{rate of clearings of } X_t^*}{\text{rate of upcrossings of } X_t} = \frac{1}{E_0T_{a+1}} \times \frac{a!}{\lambda e^{-\rho}\rho^a},$$

where T_{a+1} denotes the first hitting time

$$T_{a+1} = \min\{t : X_t = a+1\}$$

for the $M|G|\infty$ process and E_0 (and later $\Pr_0$) indicates quantities that refer to a process started at state 0 (i.e., empty). By (iii), $q(a) \le 1$. The key fact, proved in the Appendix by a different argument, is the following lemma.

LEMMA 7. *Provided* $ES \log^2 S < \infty$, *we have* $q(a) \to 1$ *as* $a \to \infty$. □

Because the $M|G|\infty$ queue regenerates at state 0, a standard argument [12] gives an exponential limit distribution for hitting times:

$$\Pr_0\{T_{a+1} > sE_0T_{a+1}\} \to e^{-s} \quad \text{as } a \to \infty, \text{ uniformly in } s.$$

This implies that the point process of clearing times of X^*, with time normalized by E_0T_{a+1}, converges (as $a \to \infty$) to a Poisson point process of rate 1. Lemma 7 now implies that the point process of upcrossings of the *stationary* queue X_t undergoes the same convergence. In particular, the (rescaled) time of the first upcrossing of X_t converges in distribution to the time of the first event of the Poisson process:

$$\Pr\{T_{a+1} > sE_0T_{a+1}\} \to e^{-s} \quad \text{as } a \to \infty, \text{ uniformly in } s \tag{3.9}$$

(which differs from the previous assertion, because it concerns the queue started with the stationary queue-size distribution rather than a queue started empty). The uniformity in (3.9) and below is a consequence of the elementary fact that, in the context of convergence of distribution functions to a continuous distribution function, pointwise convergence implies uniform convergence.

Defining $s = s(a, t_0)$ by

$$s\frac{a!}{\lambda e^{-\rho}\rho^a} = t_0,$$

we can restate (3.9) as

$$\Pr\{T_{a+1} > sE_0T_{a+1}\} - \exp(-t_0\lambda e^{-\rho}\rho^a/a!) \to 0 \quad \text{as } a \to \infty, \text{ uniformly in } t_0.$$

Now $sE_0T_{a+1} \to t_0$ by Lemma 7, so

$$\Pr\{T_{a+1} > t_0\} - \exp(-t_0\lambda e^{-\rho}\rho^a/a!) \to 0 \text{ as } a \to \infty, \text{ uniformly in } t_0.$$

But this gives (3.8a), since the events $\{T_{a+1} > t_0\}$ and $\{\max_{t\le t_0} X_t \le a\}$ are the same, provided $X_0 \le a$.

Equation (3.8b) is an immediate result of (3.8a) and the definition of a_n [cf. (3.7)], since it provides for $\Pr\{M_n > \lfloor a_n \rfloor - 1\} \to 1$ and $\Pr\{M_n < \lfloor a_n \rfloor + 2\} \to 1$. □

The rest of this subsection is devoted to the proof of Theorem 2(ii) regarding the size of HwLD under the $GI|G|\infty$ model. Our result will follow easily from the following estimate of the tail of the queue length in a $GI|G|\infty$ queue.

LEMMA 8. *In the stationary* $GI|G|\infty$ *queue, let* N *be the number of customers seen by an arriving customer. Then*

$$\Pr\{N \ge n\} = o(\alpha^n) \text{ as } n \to \infty, \text{ for every } \alpha > 0$$

provided the lifetime S *satisfies* $\Pr\{S > x\} = O(e^{-\beta x})$ *for some* $\beta > 0$.

Proof. Consider the stationary queue, conditioned on an arrival at time $\tau_0 = 0$. The previous arrivals were at times $(-\tau_1, -\tau_2, \ldots)$, where $\tau_n = \sum_{i=1}^n \xi_i$, and the ξ_i are the interarrival times. Write $G(x) = \Pr\{S \ge x\}$, where S is the lifetime. The distribution of N, the number of customers seen by the arriving customer at time 0, when the previous arrival times are given, can be described as follows:

$$\text{For } given\ (\tau_1, \tau_2, \ldots),\ N \text{ is distributed as } \sum_{i=1}^{\infty} 1_{A_i},$$
$$\text{where the } A_i \text{ are } independent \text{ and } \Pr\{A_i\} = G(\tau_i)\ .$$

(Here A_i is the event that the customer who arrived at time $-\tau_i$ is still present at time 0.)

Now fix $t_0 > 0$. Split N as $N_1' + N_2'$, where N_1' is the part of the sum $\sum 1_{A_i}$ over those i with $\tau_i \le t_0$ and where N_2' is the part of the sum $\sum 1_{A_i}$ over those i with $\tau_i > t_0$. Obviously,

$$N_1' \le N_1 \equiv \max\{n : \tau_n \le t_0\},$$

and N_2' has distribution described as follows:

$$\text{For } given \; (\tau_1, \tau_2, \ldots),\; N_2' \text{ is distributed as } \sum_{i=N_1+1}^{\infty} 1_{A_i},$$
$$\text{where the } A_i \text{ are independent with } \Pr\{A_i\} = G(\tau_i).$$

Now, the process $(\tau_{N_1+i} - t_0; i \ge 1)$ is just a delayed version of the renewal process $(\tau_i; i \ge 0)$. (*Delayed* means there is not necessarily an arrival at time 0.) By using the natural coupling between this delayed process and the undelayed renewal process and the fact that $G(\cdot)$ is decreasing, we can represent $N_2' \le N_2$, where N_2 has distribution described as follows:

$$\text{For given } (\tau_0, \tau_1, \tau_2, \ldots),\; N_2 \text{ is distributed as } \sum_{i=0}^{\infty} 1_{A_i}$$
$$\text{where the } A_i \text{ are independent with } \Pr\{A_i\} = G(t_0 + \tau_i).$$

Thus, $N \le N_1 + N_2$, and we analyze these terms separately.

We first show

$$\Pr\{N_1 \ge n\} = o(\alpha^n) \quad \text{as } n \to \infty, \text{ for every } \alpha > 0. \tag{3.10a}$$

Indeed, given α, consider K sufficiently large that $\Pr\{\xi < t_0/K\} \le \alpha/2$. Such a K exists because $\Pr\{\xi_i > 0\} = 1$. Then, as $n \to \infty$

$$\Pr\{N_1 \ge n\} = \Pr\left\{\sum_1^n \xi_i \le t_0\right\} \le \binom{n}{K} \Pr^{n-K}\{\xi \le t_0/K\} = o(2\Pr\{\xi \le t_0/K\})^n,$$

where the inequality above is a simple consequence of the fact that at most K of the ξ's can exceed t_0/K. This implies assertion (3.10a).

Now consider N_2. A standard method (see, e.g., the discussion of large deviations in [7, §1.9]) of obtaining exponentially small tail bounds on an r.v. is by studying the moment generating function. In particular, we can use the general inequality $\Pr\{X \ge a\} \le E[g(X)]/g(a)$, which holds for any nondecreasing function $g(\cdot)$. Set $g(x) = \exp(\phi x)$; then $Eg(X)$ is the moment generating function of X. The idea of the following proof is to show that $Eg(N_2) = O(1)$, and then $\Pr\{N_2 > n\} = O(\exp(-\phi(t_0)n))$ for some $\phi(t_0) \to \infty$ as $t_0 \to \infty$.

By hypothesis about the lifetime distribution, there exist $A < \infty$ and $\beta > 0$ such that

$$G(x) \le Ae^{-\beta x} \quad \text{for all } x.$$

Choose $\gamma < \infty$ such that

$$\gamma E e^{-\beta\xi} < \gamma - 1. \tag{3.10b}$$

LEMMA 9. *For all sufficiently small $\theta > 0$,*

$$E \exp\left(\theta \sum_{i=0}^{\infty} e^{-\beta \tau_i}\right) \leq \exp(\theta\gamma).$$

Proof. See the Appendix. □

Consider now $\phi > 0$. Then,

(3.10c)

$$E \exp(\phi N_2 | \tau_0, \tau_1, \ldots) = \prod_{i=0}^{\infty}(1 + (e^{\phi} - 1)G(t_0 + \tau_i)) \leq \prod_{i=0}^{\infty}(1 + (e^{\phi} - 1)Ae^{-\beta t_0}e^{-\beta \tau_i})$$
$$\leq \exp((e^{\phi} - 1)Ae^{-\beta t_0} \textstyle\sum_{i=0}^{\infty} e^{-\beta \tau_i}).$$

Now it is straightforward to find a function such that $\phi(t_0) \to \infty$ as $t_0 \to \infty$ and also such that

$$\theta(t_0) \equiv (e^{\phi(t_0)} - 1)Ae^{-\beta t_0} \to 0.$$

Taking expectations over the arrival times in (3.10c) and applying Lemma 9,

$$E \exp(\phi(t_0)N_2) \leq \exp(\theta(t_0)\gamma).$$

We have from the moment generating function approach, as $n \to \infty$,

$$\Pr\{N_2 \geq n\} = O(\exp(-\phi(t_0)n)).$$

Recall $N \leq N_1 + N_2$. Putting $\alpha = \exp(-\phi(t_0))$ in (3.10a),

$$\Pr\{N \geq 2n\} \leq \Pr\{N_1 \geq n\} + \Pr\{N_2 \geq n\} = O(\exp(-\phi(t_0)n)),$$

as $n \to \infty$. This establishes Lemma 8, because $\phi(t_0) \to \infty$. □

Returning to the proof of Theorem 2(ii), consider $U_H(t)$, where the number H of buckets is a fixed constant. By (2.2), for any $1 \leq i \leq H$ with $k \leq n$,

$$U_H(\tau_k) \leq H \cdot \max_{1\leq j\leq n} N_H^{(i)}(\tau_j^{(i)}).$$

Hence, $\Pr\{U_H(\tau_k) > n\} = o(\alpha^n)$ as well. It follows easily that $\Pr\{\max_{1\leq k\leq n} U_H(\tau_k) > a_n\} = n \cdot o(\alpha^{a_n})$, for any $\alpha > 0$. Pick $a_n = -\log_\alpha n$, for arbitrary $0 < \alpha < 1$, to find $\Pr\{\max_{1\leq k\leq n} U_H(\tau_k) > -\log_\alpha n\} = n \cdot o(1/n) = o(1)$. This proves (2.4b), since α can be arbitrary small.

3.2. Limiting excess space. We now turn our attention to the evaluation of the excess space. We first prove Theorem 3(i) for a stationary $M|G|\infty$ model of HwLD and then Theorem 3(ii) for arbitrary arrivals and lifetimes.

For an $M|G|\infty$ model of HwLD, let $U_H(\tau_k-)$ denote the table size just before the kth arrival. By PASTA, we see that $U_H(\tau_k-) - H$ is distributed as $U_H(0) - H$, which by Theorem 1(ii) has the Poisson (ρ) distribution for each k. By applying Corollary 5(i), we see

$$\Pr\left\{\max_{1\leq k\leq n} U_H(\tau_k-) \leq a_n + 1 + H\right\} \to 1, \tag{3.11}$$

where a_n satisfies (3.7). Now $N_H(t)$ is the $M|G|\infty$ queue length process, and Theorem 6 provides sharp asymptotics for the maximum queue length in such a queue. Comparing (3.11) with (3.8b), one immediately obtains (2.5a) of Theorem 3(i).

Finally, we leave the realm of queueing models to prove Theorem 3(ii), which concerns the case of arbitrary deterministic arrival and departure times. First imagine the hashing table is empty at time 0. There are arrivals at arbitrary times $0 < \tau_1 < \tau_2 \cdots < \tau_n$ with departures at arbitrary times $\eta_k > \tau_k$. Fix n. The process $N_H(t)$ and the maximum $N^* \equiv \max_{k \leq n} N_H(\tau_k)$ are deterministic. The only *probabilistic element* is the choice of bucket on arrival. We first argue that the general case can be reduced to a certain special case. Regard the arrival times and assignments to buckets as fixed, but make the following modifications. First, put N^* items in the table at time 0, but make them all depart before τ_1. Then repeat the following procedure: If there is some departure at some time $\eta < \tau_n$ that causes $N_H(\eta) = N^* - 2$, then choose the first such η and delay the departure until a time immediately after the first arrival $\tau_j > \eta$ at which $N_H(\tau_j) = N^*$. (If there is no such time τ_j, then the item stays forever.)

It is easy to show that after a finite number of such changes, there will be no such departure time η. We are then in the *special case* where $N_H(\tau_1-) = N^* - 1$ and where arrivals and departures alternate, so that $N_H(t)$ alternates between $N^* - 1$ and N^* up until time τ_n. The point is that delaying an item's departure cannot decrease any $U_H(t)$. Theorem 3(ii) concerns an upper bound for $\max_{k \leq n} U_H(\tau_k)$ in terms of N^*. Going from the general case to the special case can only increase the former quantity and leaves N^* unchanged. Therefore, it suffices to consider the special case.

Fix a time t just after an arrival, and look backwards in time from t. Let X_i be the bucket that contained the ith-from-last departing item (before t). Let Y_i be the bucket that contained the ith-from-last arriving item (before t). Write $f(i) = j$ to mean that the ith-from-last departure was the jth-from-last arrival. Then $f(i) > i$ by the alternation property. Let B_t be the number of excess items at time t. Such an item is one that was, say, the ith departure before t but has not yet been removed by subsequent arrivals. This requires

$$X_i \text{ is different from all of } Y_1, \ldots, Y_i.$$

Thus, B_t is exactly the number of i's for which this holds. The next lemma abstracts the structure of B_t. Theorem 3(ii) follows by applying this lemma to $B = U_H(\tau_k) - N^*$, summing over $k = 1, \cdots, n$, and appealing to Boole's inequality.

LEMMA 10. *Let $f : \{1, 2, 3, \ldots\} \to \{1, 2, 3, \ldots\}$ be a one-to-one function with $f(i) > i$. Let $(Y_i; i \geq 1)$ be independent, uniform on $\{1, \ldots, H\}$. Let $X_i = Y_{f(i)}$. Let A_i be the event*

$$X_i \textit{ is different from all of } Y_1, \ldots, Y_i.$$

Let B be the counting r.v. $B = \sum_{i \geq 1} 1_{A_i}$. Then, for $b > H$,

$$\Pr\{B \geq b\} \leq 2 \left(\frac{H + b}{2b} \right)^{b/2} \left(\frac{H + b}{2H} \right)^{H/2}.$$

Proof. The proof uses the following martingale-type bound, which we prove first. A good modern reference for martingales and σ fields is [7, Chap. 4]. The martingale M_n we use is the "multiplicative" analogue of the usual "additive" martingale associated with a sequence of events, the latter appearing, e.g., in [7, Thm. 4.4.10].

LEMMA 11. *Let A_i be events adapted to increasing σ fields $(\mathcal{F}_i)$, $i \geq 1$. Let $B = \sum_{i \geq 1} 1_{A_i}$. Then*

$$\Pr\{B \geq b\} \leq 2 \sqrt{\inf_{z > 1} z^{-b} E D_z},$$

where

$$D_z = \prod_{i\geq 1} E(z^{1_{A_i}}|\mathcal{F}_{i-1}) = \prod_{i\geq 1}(1 + (z-1)\Pr\{A_i|\mathcal{F}_{i-1}\}),$$

where we have used the conditional version of the expansion $Ez^{1_A} = 1 + (z-1)\Pr\{A\}$.

Proof of Lemma 11. It is enough to prove the bound for fixed $z > 1$ such that $ED_z < \infty$. Write $M_0 = 1$,

$$M_n = z^{\sum_{i=1}^n 1_{A_i}} \Big/ \prod_{i=1}^n E(z^{1_{A_i}}|\mathcal{F}_{i-1}), \qquad n \geq 1.$$

Then $\{M_n\}$ is a positive martingale. By the martingale convergence theorem [7, Cor. 4.2.11], M_n converges a.s. to some limit r.v. M_∞, and $EM_\infty \leq EM_0 = 1$. Plainly, M_∞ "ought to be" equal to z^B/D_z, and this is verified by noting that the denominator in the definition of M_n is increasing in n and converges a.s. to the a.s. finite limit D_z. Thus, we have proved

$$E(z^B/D_z) \leq 1\,. \tag{3.12}$$

Thus,

$$\begin{aligned}
\Pr\{B \geq b\} &\leq \Pr\{D_z > a\} + \Pr\{B \geq b, D_z \leq a\} \quad \text{for any } a > 0\\
&\leq ED_z/a + \Pr\{z^B/D_z \geq z^b/a\} \leq ED_z/a + a/z^b \quad \text{by (3.12)}\\
&\leq 2\sqrt{z^{-b}ED_z} \quad \text{putting } a = \sqrt{z^b ED_z}.
\end{aligned}$$

□

We evaluate now the required infimum. Let $\mathcal{F}_i$ be the σ field generated by $(Y_j;\ j \leq i+1)$ and $(X_j; j \leq i)$. Then,

$$\Pr\{A_i|\mathcal{F}_{i-1}\} = V_i/H, \qquad i \geq 1,$$

where V_i is the number of values *not* taken by $Y_1, \ldots, Y_i$. So for $z > 1$, the quantity D_z is

$$D_z = \prod_{i\geq 0}(1 + (z-1)V_i/H) \leq \exp\left(\frac{z-1}{H}\sum_{i\geq 0} V_i\right). \tag{3.13}$$

Now we can rewrite

$$\sum_{i\geq 0} V_i = \sum_{k=1}^{H} k\eta_k, \tag{3.14}$$

where η_k is the waiting time for the process V_i to go from k to $k-1$. The r.v.'s η_k are independent with (different) geometric distributions

$$\Pr\{\eta_k = u\} = (1 - k/H)^{u-1}\, k/H, \qquad u \geq 1.$$

(This is just the elementary argument for the classical coupon collector's problem with equally likely coupons.) The associated generating function may be written as

$$E\theta^{\eta_k} = \left(1 - \frac{H}{k}(1-\theta^{-1})\right)^{-1}, \qquad |\theta| < \frac{H}{H-k}. \tag{3.15}$$

Combining (3.13)–(3.15) gives

$$(ED_z)^{-1} \geq \prod_{k=1}^{H} \left(1 - \frac{H}{k}\left(1 - \exp\left(-\frac{(z-1)k}{H}\right)\right)\right).$$

Now, $1 - y^{-1}(1 - e^{-ay}) \geq 1 - a$ for $a, y > 0$, and so each term in the product is $\geq 1 - (z-1)$. This gives

$$ED_z \leq (2-z)^{-H}, \qquad 1 < z < 2,$$

and by Lemma 11 we obtain

$$\Pr\{B \geq b\} \leq 2\sqrt{\inf_{1<z<2} z^{-b}(2-z)^{-H}}.$$

Elementary calculus gives the exact infimum at $z = 2b/(b+H)$. Since the proof requires $z > 1$, the lemma only holds for $b > H$. Theorem 3(i) indicates that lower values of b are not interesting anyway. □

4. Concluding remarks. Our main results of this paper concern *hashing with lazy deletion*. In particular, we assessed the average wasted space, the (maximum) excess space, the maximum space required by HwLD, etc. Our approach in §3 can be extended to evaluate data structures, such as lists, dictionaries, stacks, priority queues, and sweepline structures for geometry and VLSI [20].

There is an important conceptual difficulty buried beneath our asymptotics. Consider again a single $M|G|\infty$ with the queue length denoted by $N(t)$ and the arrival rate by ρ. Consider the behavior of the maximum $M_T(\rho) \equiv \max_{0\leq t\leq T} N(t)$. Having in mind the application to computer storage and VLSI, it is natural to suppose that ρ is at least moderately large. Theorem 6 says

$$M_T(\rho) \sim \frac{\log T}{\log\log T} \quad \text{as } T \to \infty,\ \rho \text{ fixed}.$$

It is natural to interpret this as establishing an approximation

$$M_T(\rho) \approx \frac{\log T}{\log\log T} \quad \text{for } T \text{ large}. \tag{4.1}$$

However, substituting $T = e^\rho$ would give $M_T(\rho) \approx \rho/\log\rho$, which is absurd because, trivially, $EM_T(\rho) \geq \rho$. Thus, if $\rho = 100$, then e^{100} arrivals is not large enough for the asymptotics to be valid! [In fact, a little more analysis shows that the approximation (4.1) is valid asymptotically as $T, \rho \to \infty$ *if and only if* T increases superexponentially fast in ρ.]

For practical applications it is much more sensible to consider T as being at most polynomially large in ρ. Classical queueing theory has apparently paid no attention to polynomial-time maxima. Mathieu and Vitter [17], [18] have initiated that type of analysis for HwLD. We have also obtained some results of this nature, which we may present in a forthcoming paper. Let us mention one result for the $M|G|\infty$ queue, stronger than those in [17], [18]. Qualitatively, the idea is that for large ρ, the standardized process $Y(t) \equiv (N(t) - \rho)/\rho^{1/2}$ behaves like the standardized Ornstein–Uhlenbeck process. So, appealing to the known asymptotic behavior of the Ornstein–Uhlenbeck process, it is easy to get a heuristic approximation in the spirit of [4], [14]:

$$\Pr\{(M_T(\rho) - \rho)/\rho^{1/2} \leq b\} \approx \exp(-Tb\phi(b)), \quad b > 1, \tag{4.2}$$

where $\phi(\cdot)$ is the *Standard Normal* density function. Proving rigorously the sharp result asserted in (4.2) seems difficult for technical reasons: The usual formalization by weak convergence of processes gives this result only for $T = T(\rho) \to \infty$ slowly with ρ. On the other hand, a crude consequence of (4.2) is

$$M_T(\rho) \sim \rho + \rho^{1/2}\psi(1/T),$$

where $\psi(\cdot)$ is the inverse function of $x\phi(x), x > 1$. The first term in the expansion of $\psi(1/T)$ is $\sqrt{2\log T}$. Thus, (4.2) would imply the much weaker result: For T polynomially large in ρ,

$$M_T(\rho) \approx \rho + \rho^{1/2}(\sqrt{2\log T} + o(1)).$$

This weaker result can be proved rigorously, under some additional assumptions.

Appendix.

A. Proof of Lemma 7. Lemma 7 asserted that $q(a) \to 1$ as $a \to \infty$, where $q(a)$ is the chance that a typical upcrossing time of X_t is a clearing time of X_t^*. Call an upcrossing of X_t at t *special* if no item present at time t was present at the previous upcrossing. Clearly, a special upcrossing time is a clearing time for X_t^*, so it suffices to prove

$$q'(a) \equiv \text{ chance that a typical upcrossing is not special } \to 0 \text{ as } a \to \infty.$$

It is conceptually easier to reverse time and consider *downcrossings* from $a+1$ to a. A downcrossing at t is special if and only if all the items present at t have departed before the next downcrossing. Thus, a sufficient condition for a downcrossing, at $t = 0$ without loss of generality, to be special is that the queue length does not return to $a+1$ before the time L, at which every item present at time 0 has departed. So

$$q'(a) \le \Pr_{\downarrow}\{X_t = a+1 \text{ for some } t < L\},$$

where $\Pr_{\downarrow}$ denotes probabilities for the stationary process conditional on there being a downcrossing from $a+1$ to a at time 0.

Write

$$X_t = X_t^+ + X_t^-, \qquad t > 0,$$

where X_t^+ denotes items in storage at time t that arrived after time 0 and X_t^- denotes items in storage at time t that arrived before time 0. We see that, under $\Pr_{\downarrow}$,

(i) X_t^+ is the $M|G|\infty$ queue length process, started at 0;

(ii) $(a - X_t^-)$ is the Poisson counting process of rate $\theta(t) = \Pr\{S > t\}$, conditioned on the total number of events equaling a.

Now $q'(a)$ is further bounded by the sum of the following three probabilities.

$$\Pr_{\downarrow}\{X_t = a+1 \text{ for some } t \le t_0\}, \tag{A.1a}$$

$$\Pr\{X_t^+ > b(t) \text{ for some } t \ge t_0\}, \tag{A.1b}$$

$$\Pr_{\downarrow}\{X_t^- \ge a+1-b(t) \text{ for some } t_0 \le t < L\}, \tag{A.1c}$$

where t_0 and $b(t), t \geq 0$ are arbitrary.

Now it is easy to see that $a - X_t^- \to \infty$ as $a \to \infty$ with t fixed, and it follows that, for fixed t_0, the probability in (A.1a) tends to 0 as $a \to \infty$.

Next, if c_i is a nondecreasing integer-valued sequence with its continuous expansion $c(t) \sim 2\log t/\log\log t$, then (using easy Poisson tail estimates) $\Pr\{X_i > c_i\} = o(i^{-2+\varepsilon})$, $\varepsilon > 0$. This leads to an estimate for the *stationary* process X:

$$\Pr\{X_i \leq c_i \text{ for all sufficiently large } i\} = 1.$$

Let A_i be the number of arrivals during the interval $[i, i+1]$. Since the (A_i) are independent Poissons, Corollary 5 shows

$$\Pr\{A_i \leq c_i' \text{ for all sufficiently large } i\} = 1$$

for suitable increasing integer-valued c_i'. Because $X_t \leq X_i + A_i$ on $i \leq t \leq i+1$, we deduce

$$\Pr\{X_t > b(t) \text{ for some } t \geq t_0\} \to 0 \quad \text{as } t_0 \to \infty,$$

where $b(t) = c(\lfloor t \rfloor) + c'(\lfloor t \rfloor) \sim 3\log t/\log\log t$. (In fact, a more careful argument shows "3" could be replaced by "2.") Thus, the quantity in (A.1b) tends to 0 as $t_0 \to \infty$, because X^+ is just the $M|G|\infty$ process started empty, and so we can take $X_t^+ \leq X_t$.

We are left with the problem of bounding (A.1c): Precisely, it suffices to prove

$$\lim_{t_0\to\infty} \limsup_{a\to\infty} \Pr_\downarrow\{X_t^- \geq a + 1 - b(t) \text{ for some } t_0 \leq t < L\} = 0. \tag{A.2}$$

Since $X_t^- \geq 1$ for $t < L$, we may rewrite the probability as

$$\Pr_\downarrow\{X_t^- \geq \max(1, a + 1 - b(t)) \text{ for some } t \geq t_0\}.$$

Consider the inverse function $b^{-1}(m) = \inf\{t : b(t) \geq m\}$. Since X_t^- is decreasing in t, it suffices to check the inequality for t of the form $b^{-1}(m)$, and the preceding probability becomes

$$\Pr_\downarrow\{X_{b^{-1}(m)}^- \geq a + 1 - m \text{ for some } b(t_0) \leq m \leq a\} \leq \sum_{m=b(t_0)}^{a} \frac{E_\downarrow X_{b^{-1}(m)}^-}{a+1-m}.$$

We now quote a standard fact. Let N_t be a Poisson counting process on $0 \leq t < \infty$ with rate $\theta(t)$ and such that $\int_0^\infty \theta(s)ds < \infty$. Then,

$$E(N_t | N_\infty = a) = a \int_0^t \theta(s)ds \Big/ \int_0^\infty \theta(s)ds.$$

This follows from the fact [19, exercise 2.24a] that, conditional on $N_\infty = a$, the positions of these a points are distributed as the positions of a points chosen independent from the distribution with distribution function $F(t) = \int_0^t \theta(s)ds / \int_0^\infty \theta(s)ds$. By (ii) above, we can apply this fact to $N_t = a - X_t^-$ to get

$$\begin{aligned} E_\downarrow X_t^- &= a - a \int_0^t \theta(s)ds \Big/ \int_0^\infty \theta(s)ds \\ &= a \int_t^\infty \theta(s)ds \Big/ \int_0^\infty \theta(s)ds = aE(S-t)^+/ES. \end{aligned}$$

Thus, the proof of (A.2) reduces to the proof of

$$\lim_{t_0\to\infty} \limsup_{a\to\infty} \sum_{m=b(t_0)}^{a} \frac{a}{a+1-m} E(S - b^{-1}(m))^+ = 0.$$

Splitting the sum at $a/2$, we see it is enough to prove

$$\sum_{m=1}^{\infty} E(S - b^{-1}(m))^+ < \infty,$$

$$a \log a \;\; E(S - b^{-1}(a/2))^+ \to 0 \quad \text{as } a \to \infty.$$

But these are simple consequences of the fact

$$\log b^{-1}(m) \sim \tfrac{1}{3} m \log m,$$

together with the inequality

$$E(S-c)^+ \le \frac{ES \log^2 S}{\log^2 c}, \; c > 1.$$

This completes the proof of Lemma 7. □

B. Proof of Lemma 9. Lemma 9 asserted that for all sufficiently small $\theta > 0$,

$$E \exp\left(\theta \sum_{i=0}^{\infty} e^{-\beta \tau_i}\right) \le \exp(\theta\gamma).$$

Write $Q_n = \sum_{i=0}^{n} \exp(-\beta\tau_i)$. Then we have the recursion

$$Q_{n+1} \stackrel{d}{=} 1 + e^{-\beta\xi} Q_n,$$

in which ξ and Q_n are taken independent. Consider some $\theta_0 > 0$. If we can show, by induction on n, that

$$E \exp(\theta Q_n) \le \exp(\theta\gamma) \quad \text{for all } 0 \le \theta \le \theta_0,$$

then the Lemma 9 follows by letting $n \to \infty$. If the above holds for n, then

$$\begin{aligned} E \exp(\theta Q_{n+1}) &= e^{\theta} E \exp(\theta e^{-\beta\xi} Q_n) \quad \text{by our recurrence} \\ &\le e^{\theta} E \exp(\theta e^{-\beta\xi} \gamma) \quad \text{by induction assumption.} \end{aligned}$$

To make the induction go through, we require this to be bounded by $\exp(\theta\gamma)$, and this rearranges to the requirement

$$E \exp(\theta(e^{-\beta\xi}\gamma - (\gamma - 1))) \le 1, \qquad 0 \le \theta \le \theta_0. \tag{A.3}$$

But this fact (for some θ_0) follows from the choice of γ in relation (3.10b), which implies that the derivative at $\theta = 0$ of the left-hand side of (A.3) is negative. □

REFERENCES

[1] M. ABRAMOWITZ AND I. STEGUN, *Handbook of Mathematical Functions*, Dover, New York, 1964.

[2] O. AVEN, E. G. COFFMAN, JR., AND Y. A. KOGAN, *Stochastic Analysis of Computer Storage*, D. Reidel, Boston, 1987.

[3] A. AHO, J. HOPCROFT, AND J. ULLMAN, *Data Structures and Algorithms*, Addison-Wesley, Reading, MA, 1983.

[4] D. ALDOUS, *Probability Approximations via the Poisson Clumping Heuristic*, Springer-Verlag, Berlin, New York, 1989.

[5] C. W. ANDERSON, *Extreme value theory for a class of discrete distributions with applications to some stochastic processes*, J. Appl. Probab., 7 (1970), pp. 99–113.

[6] S. BERMAN, *Limiting distribution of the maximum term in sequences of dependent random variables*, Ann. Math. Statist., 33 (1962), pp. 894–908.

[7] R. DURRETT, *Probability: Theory and Examples*, Wadsworth, Belmont, CA, 1991.

[8] W. FELLER, *An Introduction to Probability Theory and its Applications*, Vol. II, John Wiley, New York, 1971.

[9] P. FLAJOLET, J. FRANCON, AND J. VUILLEMIN, *Sequence of operations analysis for dynamic data structures*, J. Algorithms, 1 (1980), pp. 111–141.

[10] J. GALAMBOS, *The Asymptotic Theory of Extreme Order Statistics*, John Wiley, New York, 1978.

[11] N. KAPLAN, *Limit theorems for a $GI/G/\infty$ queue*, Ann. Probab., 3 (1975), pp. 780–789.

[12] J. KEILSON, *A limit theorem for passage times in ergodic regenerative processes*, Ann. Math. Statist., 37 (1966), pp. 866–870.

[13] L. KLEINROCK, *Queueing Systems*, Vol. 1, John Wiley, New York, 1976.

[14] G. LOUCHARD, C. KENYON, AND R. SCHOTT, *Data Structures Maxima*, Fund. Comp. Theory, FCT '91, Berlin Springer Lecture Notes in Computer Science, 529, pp. 339–349.

[15] G. LOUCHARD, B. RANDRIANARIMANANA, AND R. SCHOTT, *Dynamic algorithms in D.E. Knuth's model: A probabilistic analysis*, Theoret. Comput. Sci., 93 (1992), pp. 201–225.

[16] J. MORRISON, L. SHEPP, AND C. VAN WYK, *A queueing analysis of hashing with lazy deletion*, SIAM J. Comput., 16 (1987), pp. 1155–1164.

[17] C. MATHIEU AND J. S. VITTER, *The maximum size of dynamic data structures*, SIAM J. Comput., 20 (1991), pp. 807–823.

[18] ———, *General methods for the analysis of the maximum size of dynamic data structures*, in Proc. 16th International Colloquium on Automata, Languages, and Programming, Stresa, Italy, 1989, pp. 473–487.

[19] S. M. ROSS, *Stochastic Processes*, John Wiley, New York, 1983.

[20] T. G. SZYMANSKI AND C. VAN WYK, *Space efficient algorithms for VLSI artwork analysis*, in Proc. 20th IEEE Design Automation Conference, 1983, pp. 734–739.

[21] W. SZPANKOWSKI, *On the maximum queue length with applications to data structure analysis*, in Proc. 27th Annual Allerton Conference on Communication, Control and Computing, Univ. of Illinois, Urbana-Champaign, IL, 1989, pp. 263–272.

[22] C. VAN WYK AND J. S. VITTER, *The complexity of hashing with lazy deletion*, Algorithmica, 1 (1986), pp. 17–29.

[23] W. WOLFF, *Stochastic Modeling and the Theory of Queues*, Prentice-Hall, Englewood Cliffs, NJ, 1989.

SIAM J. COMPUT.
Vol. 21, No. 4, pp. 733–742, August 1992

© 1992 Society for Industrial and Applied Mathematics
007

COMPLETE PROBLEMS AND STRONG POLYNOMIAL REDUCIBILITIES*

K. GANESAN† AND STEVEN HOMER‡

Abstract. A set A is m-reducible to a set B if and only if there is a polynomial-time computable function f such that for all x, $x \in A \Leftrightarrow f(x) \in B$. A set C is m-complete for a class S if $C \in S$ and all sets in S are m-reducible to C. One-reducibility and one-completeness can be defined by requiring f to be one–one. Two sets A and B are p-isomorphic if the function f can be taken one-to-one, onto, and polynomially invertible. In this paper it is shown that all the m-complete sets are one–one complete for DTIME($2^{\mathcal{O}(n)}$), NTIME($2^{\mathcal{O}(n)}$), and the class of recursively enumerable sets. Further, all the sets complete for NTIME($2^{\mathcal{O}(n)}$) under 1–L (or two-way DFA) reductions are p-isomorphic. All the m-complete sets for DTIME($2^{\mathcal{O}(n)}$) are p-isomorphic if and only if all the m-complete sets for DTIME(2^{poly}) are p-isomorphic.

Key words. complexity classes, polynomial reductions, completeness

AMS(MOS) subject classifications. 68Q15, 03D15, 03D25

1. Introduction. This paper studies intrinsic properties of complete problems for a variety of complexity classes. Complete problems are central to complexity theory and are ubiquitous in both the theory and the important examples. The most important and most thoroughly studied complete sets are the NP-complete problems. Their study makes up a major body of research in concrete complexity theory, one aspect of which has concerned the structural properties of these sets. In particular, the isomorphism conjecture of Berman and Hartmanis [BH77], which states that all the NP-complete sets are polynomial-time isomorphic, has been important in this study. It has provided the impetus for a large body of research in structural complexity theory over the past decade.

Although the conjecture is still far from being settled, there have recently been a number of results exploring some of its generalizations and extensions. They can be interpreted as indicating what the eventual outcome of the conjecture may be and what methods may be used to attack it. The generalizations bear upon the structure of complete sets for other complexity classes and for different effective reducibilities. Recent important work in this area includes that of Ko, Long, and Du [KLD87] relating the existence of one-way functions to the isomorphism conjecture for exponential time complete sets; the research of Kurtz, Mahaney, and Royer [KMR88] on generalizations of the isomorphism conjecture to larger complexity classes; and the work of Joseph and Young [JY85] and Watanabe [Wat85] on the structure of possible counter-examples to the isomorphism conjecture and its generalizations.

This paper contributes to several aspects of this study. First, we compare complete sets for the most common strong polynomial-time reducibilities, polynomial many–one ($\leq_m^p$) and polynomial one–one ($\leq_1^p$) reducibility. We carry this out for a number of well-studied complexity classes. Second, we consider the consequences of these results for the isomorphism conjecture for several often studied complexity classes as well as for the class of recursively enumerable (RE) sets. Our approach comes from the well-known result of Myhill [Rog67] stating that all the RE complete sets are recursively isomorphic. This result was the progenitor of the research of Berman and Hartmanis and of their

*Received by the editors October 16, 1989; accepted for publication July 25, 1991. This work was supported in part by National Security Agency grant MDA904-87-H-2003 and by National Science Foundation grant MIP-8608137. A preliminary version of this paper was presented at the 1989 Symposium on Theoretical Aspects of Computer Science, Paderborn, West Germany.

†Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, Florida 33431.

‡Computer Science Department, Boston University, Boston, Massachusetts 02215.

conjecture. Myhill's Theorem is proved in two parts. All the RE many–one complete sets are shown to be one–one complete. Then, all the one-one complete sets are proved to be recursively isomorphic.

Turning to the polynomial-time analog for NP sets, the equivalence between many–one and one–one completeness for polynomial-time reducibility is not known. For deterministic linear exponential-time (E) complete sets, Berman [Ber77] proved that any polynomial-time many–one complete set is polynomial-time one–one complete. In §3, we present a new and extremely simple proof of this theorem. We also prove a "transfer theorem" that says that the isomorphism conjecture holds for E if and only if it holds for DTIME(2^{poly}). This is the first result of this kind, and it would be most interesting if similar results could be proved for nondeterministic classes as well.

This same question concerning the strong polynomial reducibilities has been open for all nondeterministic classes and, in particular, for complete sets for nondeterministic linear exponential-time (NE)class. In §4, we settle this question by showing that any NE many–one complete set is also one–one complete. Our theorem applies to larger nondeterministic complexity classes and a similar proof applies to RE sets that are complete with respect to many–one polynomial-time reducibility (a result originally due to Dowd [Dow82]).

As we work toward the possible polynomial-time isomorphism of complete sets for these classes, the next question is whether these one–one reductions can be made length increasing or at least polynomially honest. Intuitively, a reduction is polynomially honest if the size of the output is within a polynomial of the size of the input (a precise definition is given in the next section). A simple lemma proves that if there is a polynomially honest reduction then there is one that is length increasing. Berman's previously mentioned result showed that for E, all complete sets are one–one length increasing equivalent. We do not have such a result for NE or other nondeterministic classes, or for RE, but we do show that the reduction we construct can be forced to be exponentially honest. Polynomial honesty remains an interesting open question.

Returning now to isomorphism questions, we exhibit some progress. We show that our results for NE apply to some limited classes of complete sets and prove polynomial isomorphism for these classes. For effective reducibilities there are very few results proving the isomorphism of all complete sets. The best known results of this type were produced by Allender [All88]. He proved that all sets complete for deterministic linear exponential class under one-way logspace (or under two-way DFA) reductions are polynomial-time isomorphic. We can apply our proof techniques for NE to show that for the same reductions, complete sets for NE (or many other larger nondeterministic time classes) are polynomial-time isomorphic. These are the first nondeterministic complete classes for which isomorphism is known.

2. Preliminaries. We assume that the reader is familiar with the usual notions of Turing machines and complexity classes, such as P, NP, E, and NE. For further details refer to [HU79].

We will consider languages to be subsets of Σ^*, where $\Sigma = \{0,1\}$. For a string $w \in \Sigma^*$, let $|w|$ be its length. We assume the standard total ordering $<$ on strings of Σ^* such that strings of smaller size precede strings of larger size, and strings of the same length are ordered lexicographically. For a set A, let $\bar{A}$ denote the complement of A. Let $\mathcal{N}$ denote the set of natural numbers. We do not distinguish between Σ^* and $\mathcal{N}$ as there is a bijection between them. For all x and y, let $\langle x,y\rangle = (x+y)(x+y+1)/2+x$. $\langle .,. \rangle$ is a standard pairing function that maps $\mathcal{N} \times \mathcal{N}$ onto $\mathcal{N}$. Note that $|\langle x,y\rangle|$ is $\mathcal{O}(|x|+|y|)$.

A set A is *polynomial-time many–one reducible* to a set B (denoted $A \leq_m^p B$) if and

only if there exists a polynomial-time computable function f such that for all x, $x \in A \Leftrightarrow f(x) \in B$. A and B are *polynomial-time many–one equivalent* (denoted $A \equiv_m^p B$) if and only if $A \leq_m^p B$ and $B \leq_m^p A$. An *m-degree* is an equivalence class under the relation $\equiv_m^p$. A set C is *m-complete* for a class of sets S if and only if $C \in S$ and every set in S is polynomial-time many–one reducible to C.

A set A is *polynomial-time one–one reducible* to a set B (denoted $A \leq_1^p B$) if and only if there exists a polynomial-time computable one–one function f such that for all x, $x \in A \Leftrightarrow f(x) \in B$. We say that A is *polynomial-time one–one, length-increasing reducible* to a set B (denoted $A \leq_{1-li}^p B$) if $A \leq_1^p B$ and for all x, $|f(x)| > |x|$. Two sets A and B are *one–one(li) equivalent* if either $A = B$ or each one is reducible to the other by a one–one(li) reduction. A one–one(li) degree is an equivalence class under the relation of one–one(li) equivalence. A set C is *one–one-complete* for a class of sets S if and only if $C \in S$ and every set in S is one–one reducible to C. A set C is *one–one-li-complete* for a class of sets S if and only if $C \in S$ and every set in S is one–one, length-increasing reducible to C. All reductions studied in this paper are computable in polynomial time unless mentioned otherwise.

A function f is *polynomially honest* if and only if for some polynomial g, for all x, $g(|f(x)|) \geq |x|$. A function f is said to be *exponentially honest* if for all x, $2^{|f(x)|} \geq |x|$.

A set A in some complexity class S is said to be *polynomially* (*exponentially*) *honest complete* if every set in S is reducible to A by a many–one polynomially (exponentially) honest function f.

A function f is said to be *strongly invertible* if there exists a polynomial-time Turing machine that on input y in the range of f prints out all the elements x such that $f(x) = y$.

A function f is *p-invertible* if and only if there exists a polynomial-time computable function g such that, for all x, $g(f(x)) = x$. A function f is a p-isomorphism if and only if f is computable in polynomial time, one–one, onto, and p-invertible. We say A and B are polynomial-time isomorphic (denoted $A \cong^p B$) if and only if for some p-isomorphism f, $f(A) = B$.

A polynomial-time computable function p is said to be a *padding function* for a set A if $(\forall x, y)\, x \in A \Leftrightarrow p(\langle x, y\rangle) \in A$. The padding function is said to be *invertible* if there exists a polynomial-time computable function q such that $\forall x, y\, q(p(\langle x, y\rangle)) = \langle x, y\rangle$.

A function f is a *one-way* function if and only if f is computable in polynomial time, f is one–one, is polynomially honest, and is not polynomial-time invertible. We define

$$\begin{aligned} \mathrm{E} &= \cup_c \mathrm{TIME}[2^{cn}], \\ \mathrm{NE} &= \cup_c \mathrm{NTIME}[2^{cn}], \\ \mathrm{DTIME}(2^{\mathrm{poly}}) &= \cup_c \mathrm{TIME}[2^{n^c}], \\ \mathrm{NTIME}(2^{\mathrm{poly}}) &= \cup_c \mathrm{NTIME}[2^{n^c}]. \end{aligned}$$

All of these classes have complete sets. In fact, they have a one–one, length increasing, invertible complete set that has a one–one, length increasing and invertible padding function. A straightforward padding argument shows that a set that is m-complete for E is also m-complete for DTIME(2^{poly}) and a set that is m-complete for NE is also m-complete for NTIME(2^{poly}). In addition to the above notations, we will use PF to denote the class of functions computable in polynomial time.

Let $\langle \phi_i \rangle_{i \in N}$ be an acceptable numbering of the partial recursive functions based on a coding of deterministic, multiple-tape Turing machines. By standard results in the literature, there is a Kleene function $T = \lambda i, x, n.[\phi_i(x)$ if Turing machine i on input x

halts within n steps; 0 otherwise] that is computable in time polynomial in $|i|, |x|$, and n. For each i, let $f_i = \lambda x.[T(j, x, k \cdot |x|^{\log\log(k)})]$, where $i = \langle j, k\rangle$. Clearly, for each i, $f_i \in$ PF and every function in PF appears in this enumeration. Thus, it follows that $\langle f_i\rangle_{i\in N}$ is an enumeration of PF. It is not difficult to see that $f_i(x)$ can also be computed in time $2^{\mathcal{O}(|i|+|x|)}$ as mentioned in [KMR88].

PROPOSITION 1. *The function* $\lambda i, x.T(j, x, k \cdot |x|^{\log\log(k)})$ *is computable in time* $2^{\mathcal{O}((|i|+\log(|x|))^2)}$.

Proof. First, note that given i and x, the predominant complexity comes from the running time of the machine j. Clearly, $|k| < |i|$ and hence $k < 2^{|i|}$. Let $z = k \cdot |x|^{\log\log(k)}$. Then, $z \leq 2^{|i|} \cdot |x|^{\log(|i|)}$. Thus, $\log(z) \leq |i| + \log(|i|) \cdot \log(|x|) \leq |i| + |i| \cdot \log(|x|) \leq (|i| + \log(|x|))^2$. Hence, $z \leq 2^{(|i|+\log(|x|))^2}$. Given i and x, we can compute j and k in polynomial time. The index of the machine that computes the function T is fixed. The running time of T is bounded by $2^{\mathcal{O}((|i|+\log(|x|))^2)}$. Since simulation can be done in square of the running time, $f_i(x)$ can be effectively computed from i and x in $2^{\mathcal{O}((|i|+\log(|x|))^2)}$ time.

3. Deterministic exponential time sets. In his Ph.D. thesis, Berman [Ber77] proved that all the m-complete sets for E are one–one, length-increasing complete. A more recent, simpler proof has been given by Watanabe [Wat85]. Our methods yield a still more direct construction that applies to $\bigcup_c \mathrm{TIME}(2^{c\cdot n^{1/k}})$ for any $k > 0$, E, and larger classes that are closed under complementation. For simplicity we will present the result only for E. Its proof points the way to the new results for nondeterministic classes presented in the next section.

THEOREM 1. *All m-complete sets for* E *are one–one-li equivalent.*

Proof. Let A be any given m-complete set in E and let K be any other arbitrary set in E. It is enough to show that $K \leq^p_{1-li} A$.

Fix an enumeration $\langle f_i\rangle_{i\in N}$ of PF functions such that $\lambda i, x.f_i(x)$ is computable in time $2^{\mathcal{O}(|i|+|x|)}$.

We construct a set M in E such that a reduction, say, f_j from M to A, is one–one-li on $\{j\} \times N$. In addition, the set M is constructed so that the function $g(x) = \langle j, x\rangle$ will be a reduction from K to M. The required one–one-li reduction from K to A is then $f(x) = f_j(g(x))$.

The following program describes the set M.

(1) **input** $\langle i, x\rangle$
(2) **if** $|f_i(\langle i, x\rangle)| \leq |\langle i, x\rangle|$
(3) **then** accept $\langle i, x\rangle$ if and only if $f_i(\langle i, x\rangle) \notin A$.
(4) **else if** $\exists\, y < x$ such that $f_i(\langle i, x\rangle) = f_i(\langle i, y\rangle)$
(5) **then** accept $\langle i, x\rangle$ if and only if $y \notin K$.
(6) **else** accept $\langle i, x\rangle$ if and only if $x \in K$.

LEMMA 1. *M is in* E.

Proof. One simply checks that the time bound for the program M on input $\langle i, x\rangle$ is $\leq 2^{c\cdot|\langle i, x\rangle|}$ for some constant c.

LEMMA 2. *If f_j is a reduction from M to A, then f_j is one–one-li on $\{j\} \times N$. Moreover, $g(x) = \langle j, x\rangle$ is a reduction from K to M.*

Proof. If f_j is not length increasing, it is not a reduction from M to A because of Step 3 of the algorithm. Therefore, f_j has to be length increasing. Suppose f_j is not one–one. Let x_2 be the least element such that for some $x_1 < x_2$, $f_j(\langle j, x_2\rangle) = f_j(\langle j, x_1\rangle)$. By the definition of M, $\langle j, x_1\rangle \in M \Leftrightarrow x_1 \in K$ and $\langle j, x_2\rangle \in M \Leftrightarrow x_1 \notin K$. Thus, f_j

cannot be a reduction from M to A, which is a contradiction. Hence, f_j is one–one-li on $\{j\} \times N$. Note that $\langle j, x\rangle \in M$ if and only if $x \in K$ from the way M is defined. The elements of the form $\langle j, x\rangle$ will always fall in Step 6 of the algorithm. Hence, $g(x) = \langle j, x\rangle$ is a reduction from K to M.

LEMMA 3. $K \leq^p_{1-li} A$.

Proof. Define $f(x) = f_j(g(x))$. Clearly, g is one–one-li. Since f_j is one–one-li on the range of g, f is one–one-li. Further, f is computable in polynomial time. It is immediate that f is a reduction from K to A.

As mentioned earlier, the above theorem holds for $\cup_c \text{TIME}(2^{c \cdot n^{1/k}})$ for any k. This is because of the fact that any set A that is m-complete for this class is also m-complete for DTIME(2^{poly}). Hence, there would be a reduction from the set M constructed in the theorem to the set A. Note that it is sufficient for the set M constructed to be in DTIME(2^{poly}).

Berman and Hartmanis [BH77] have shown that any two sets that are one–one, length increasing, and invertible equivalent are p-isomorphic. Since all many–one complete sets for E are one–one, length increasing complete, they are all p-isomorphic to each other unless one-way functions exist.

Although we cannot settle the isomorphism question for exponential-time classes, we prove that the answer is the same for E and DTIME(2^{poly}). Our proof here uses Theorem 1.

THEOREM 2. *All m-complete sets for* DTIME(2^{poly}) *are p-isomorphic if and only if all m-complete sets for* E *are p-isomorphic.*

Proof. $\Rightarrow$: This is immediate from the fact that every m-complete set for E is also m-complete for DTIME(2^{poly}).

$\Leftarrow$: Suppose that not all of the m-complete sets for DTIME(2^{poly}) are p-isomorphic. We will show that not all m-complete sets for E are p-isomorphic. Let K be any one–one, length increasing, invertible complete set for E that has a one–one, length-increasing padding function p. Since any set m-complete for E is also m-complete for DTIME(2^{poly}), K is m-complete for DTIME(2^{poly}). Let X be an m-complete set for DTIME(2^{poly}) such that $K \not\cong^p X$. Let f be any one–one-li reduction from K to X. If f is invertible in polynomial-time, then K and X would be p-isomorphic. Hence, f has to be one-way. Since K and X are not p-isomorphic, $\forall g$ one–one-li, invertible, either $g(K) - f(K)$ is an infinite set or $g(\bar{K}) - f(\bar{K})$ is an infinite set. (Proof: Suppose not. Let k be the length of the largest element in $(g(K) - f(K)) \cup (g(\bar{(K)}) - f(\bar{(K)}))$. Then, $g'(x) = g(S(x, 0^k))$ is a one–one, length-increasing, and invertible reduction from K to X. This would imply that $K \cong^p X$.)

Next, we construct a set M m-complete for E such that $K \not\cong^p M$ thus proving the theorem. The following program describes the set M.

(1) **input** y.
(2) **if** $f(x) = y$ for some $x < y$
(3) **then** accept y if and only if $x \in K$. {This makes M m-complete}
(4) **else**
(5)   **if** $\exists\, i < |y|$ such that i is the least index that is not yet canceled and
(6) $f_i(z) = y$ for some $z < y$ {Note that f cannot satisfy this condition}
(7)   **then** accept y if and only if $z \notin K$ and cancel i
(8)   **else** reject y

We claim that there is no one–one-li,invertible function g that reduces K to M. First note that $f(K) \subseteq M$ and $f(\bar{K}) \subseteq \bar{M}$. This immediately cancels each function g such that $g(K) - f(K)$ spills into $f(\bar{K})$ or such that $g(\bar{K}) - f(\bar{K})$ spills into $f(K)$. Thus, the

only one–one-li,invertible functions left to cancel are those that have an infinite number of "free" strings in at least one of $g(K) - f(K)$ or $g(\bar{K}) - f(\bar{K})$. But, these functions would be eventually canceled in Step 7 of the above algorithm. Thus, it is clear that every index corresponding to a 1-1,li, invertible function which is a potential reduction from K to M, will eventually get canceled because of the choice of f. Let $|y| = n$. Clearly, Steps 2 and 3 can be computed in time $2^{\mathcal{O}(n)}$. For Steps 5–8, we need to store the list of indices of length $\leq n$ that have been satisfied during the previous stages. The space needed for storing this list is bounded by $2^{\mathcal{O}(n)}$. The time needed to simulate all the previous stages and construct this list is also bounded by $2^{\mathcal{O}(n)}$ as there are only $2^{\mathcal{O}(n)}$ previous stages and each stage needs time at most $2^{\mathcal{O}(n)}$. Hence, the set constructed is in E.

Even though the above result is stated for E and DTIME(2^{poly}), it holds for higher deterministic classes as well.

4. Nondeterministic exponential time sets. Attention is now turned to nondeterministic subrecursive classes. First, we consider the nondeterministic class NE and show that many–one complete sets for this class are one–one complete. This result is new and settles a longstanding open problem. Our method is essentially the same but somewhat more delicate because the set M constructed in the proof must itself be in NTIME(2^{poly}). In fact, our method will apply to any nondeterministic time class larger than NE and also to $\bigcup_c \text{NTIME}(2^{c \cdot n^{1/k}})$ for any $k > 0$.

THEOREM 3. *All m-complete sets for* NE *are one–one exponentially honest equivalent.*

Proof. Let A be any given m-complete set for NE and let K be any arbitrary set in NE. It is sufficient to show that K is one–one exponentially honest reducible to A.

Let $f_1, f_2, \ldots$ be an enumeration of all polynomial-time computable functions, such that $\lambda i, x. f_i(x)$ can be computed in time $2^{\mathcal{O}((|i|+\log(|x|))^2)}$ as in Proposition 1.

Let M be the set of pairs accepted by the following procedure.

(1) **input** $\langle i, x\rangle$
(2) **if** $2^{|f_i(\langle i, x\rangle)|} < |\langle i, x\rangle|$
(3) **then** accept $\langle i, x\rangle$ if and only if $f_i(\langle i, x\rangle) \notin A$
(4) **if** $\exists\, y < x$ such that $f_i(\langle i, y\rangle) = f_i(\langle i, x\rangle)$
(5) **then if** $2^{|y|} \leq |x|$ then accept $\langle i, x\rangle$ if and only if $y \notin K$.
(6) **else** reject $\langle i, x\rangle$.
(7) **else** accept $\langle i, x\rangle$ if either (a) or (b) holds.
(8) (a) $\exists\, y > x$ such that $|y| < 2^{|x|}$ and $f_i(\langle i, x\rangle) = f_i(\langle i, y\rangle)$
(9) (b) $x \in K$

LEMMA 4. *M is in* NTIME(2^{poly}).

Proof. Let c and d be constants such that the running time of K is bounded by $2^{c \cdot |x|}$ and the running time of A is bounded by $2^{d \cdot |x|}$. Let us compute the time required for M on input $\langle i, x\rangle$. Note that computing $f_i(\langle i, x\rangle)$ takes time at most $2^{\mathcal{O}((|i|+\log(|\langle i, x\rangle|))^2)}$, which is $2^{\mathcal{O}((|\langle i, x\rangle|)^2)}$. If $2^{|f_i(\langle i, x\rangle)|} < |\langle i, x\rangle|$, the membership of $f_i(\langle i, x\rangle)$ in A can be decided deterministically in time $2^{\mathcal{O}((\langle i, x\rangle)^d)}$. Since there are only $2^{\mathcal{O}(|x|)}$ strings y that are lexicographically less than x, the condition in Step 4 of the algorithm can be computed in time $2^{\mathcal{O}(|\langle i, x\rangle|)}$. If $2^{|y|} \leq |x|$, we can decide if $y \in K$ deterministically in time $2^{2^{c \cdot |y|}} < 2^{|x|^c}$. Thus, Steps 5 and 6 can be done in time $2^{\mathcal{O}(|x|^c)}$. In Step 8, we can guess a string y that is at most of length $2^{|x|}$ and compute $f_i(\langle i, y\rangle)$ in time $2^{\mathcal{O}(|i|+\log(|\langle i, y\rangle|)^2)}$, which is $2^{\mathcal{O}((|i|+|x|)^2)}$. Thus the set M is in NTIME(2^{poly}).

Any m-complete set for NE is also m-complete for NTIME(2^{poly}), so there is a reduction from M to A.

LEMMA 5. *Let f_j be any reduction from M to A. Then $f(x) = f_j(\langle j, x\rangle)$ is a one–one exponentially honest reduction from K to A.*

Proof. It is clear from the definition that the reduction f_j has to be exponentially honest, otherwise it cannot be a reduction because of diagonalization in Steps 2 and 3. Assume f is not one–one. Let x_2 be the least element such that for some $x_1 < x_2$, $f(x_1) = f(x_2)$. There are two cases.

Case 1. $2^{|x_1|} \leq |x_2|$. In this case, $\langle j, x_1\rangle \in M$ if and only if $x_1 \in K$. $\langle j, x_2\rangle \in M$ if and only if $x_1 \notin K$. So, f_j cannot be a reduction from M to A. Contradiction.

Case 2. $2^{|x_1|} > |x_2|$. In this case, $\langle j, x_1\rangle \in M$ and $\langle j, x_2\rangle \notin M$. Again, f_j cannot be a reduction from M to A. Contradiction.

Hence, f is one–one. Having proved that f is one–one, we can easily verify that f is a reduction from K to A.

The next proposition shows that any significant improvement to the exponential honesty would imply the existence of length-increasing reductions. It is not new, but it is a useful general fact (see [You83]). It is stated here for NE but applies more generally to many other classes, including NP and RE.

PROPOSITION 2. *Any one–one polynomially honest complete* NE *set is also one–one, length-increasing complete.*

Proof. Let A be any one–one, polynomially honest complete NE set. Let K be any standard one–one, length-increasing complete set for NE that has a one–one, length-increasing padding function p. Suppose f is a polynomially honest reduction from K to A. Since f is honest, there exists some k such that $\forall\, x$, $|f(x)|^k > |x|$. Define $d(x) = x10^{|x|^k}$. Then, d is one–one and computable in polynomial-time. Also, $|d(x)| > |x|^k$. We now define $h(x) = f(p(\langle x, d(x)\rangle))$, which is the required one–one, length-increasing reduction from K to A. The function h is length-increasing since,

$$|h(x)| = |f(p(\langle x, d(x)\rangle))| > |p(\langle x, d(x)\rangle)|^{1/k} > |d(x)|^{1/k} > \left(|x|^k\right)^{1/k} = |x|.$$

h is clearly one–one and polynomial-time computable.

THEOREM 4. *If* P = UP, *then all m-complete sets for* E *are p-isomorphic. If* P = NP, *then all m-complete sets for* NE *are p-isomorphic.*

Proof. If P = UP, then one-way functions do not exist. So, all m-complete sets for E are one–one-li,invertible equivalent and hence p-isomorphic. If P = NP, then E = NE. Also, one-way functions do not exist. So, all m-complete sets for E = NE are one–one, length-increasing, invertible equivalent and hence p-isomorphic.

Next, we consider the question of what natural subclasses of the NE-complete sets include only sets p-isomorphic to standard m-complete sets. Allender [All88] has studied 1–L [HIM78] and two-way DFA [Sto74] reductions for various complexity classes. In particular, he has proven that all sets m-complete for E under 1–L (two-way DFA) reductions are p-isomorphic. These questions were left open for the class NE. We will answer these questions affirmatively for NE.

DEFINITION 1. A function f is said to be in the class 1–L of functions if f can be computed by a Turing machine with a one-way read only input tape with end markers, a one-way write only output tape and a read-write work tape which is initially set to all blanks such that for all inputs x, at most $\lceil \log |x| \rceil$ cells of the work tape are ever visited.

DEFINITION 2. A function f is said to be in the class two-way DFA of functions if f can be computed by a DFA that has a two-way input tape with end markers and a one-way left-to-right write-only output tape.

THEOREM 5. *All sets complete for* NE *under* 1–L *reductions are p-isomorphic.*

Proof. Let A be any complete set for NE under 1–L reductions. Allender [All88] has shown that all sets complete for NE under 1–L reductions are complete under length-increasing, strongly invertible many–one polynomial time reductions. Given a 1–L complete set A, we can construct a set M in NTIME(2^{poly}), as in the proof of Theorem 3, such that any reduction from M to A is one–one on $\{j\} \times \mathcal{N}$. Since A is complete NE under length-increasing, strongly invertible many–one reductions, A is also complete for NTIME(2^{poly}). Hence, there exists a many–one, length-increasing, and strongly invertible polynomial reduction from M to A. Following the ideas of the proof of Theorem 3, we can then obtain a one–one, length-increasing, and invertible reduction from K to A. Thus A is one–one-li,invertible complete and hence $K \cong^p A$.

THEOREM 6. *All sets complete for* NE *under two-way* DFA *reductions are p-isomorphic.*

Proof. First, note that the set of two-way DFA functions is a subset of linear-time computable functions. Also, two-way DFA reductions that are one–one can be proved to be linear honest. This can be proved using "crossing sequence" argument. Let f be any one-one function that is computable by a two-way DFA with q states. Suppose, f is not linear honest, then for some input x, $q(q+1)|f(x)| < |x|$. Since the length of the output is so small compared to the length of the input, we can find two positions b and c such that there is no output produced when the machine is visiting any position between b and c and the crossing sequence at b is same as the crossing sequence at c. Let $x = uvw$ where v is the substring appearing between positions b and c. Then, it is clear that $f(x) = f(uvvw) = f(uvvvw) = \ldots$. Thus f is not one–one. Further, it is known that every honest two-way DFA computable function is polynomial-time invertible [All85, p. 54].

The proof is similar to the proof of Theorem 3. We can delete Steps 2 and 3 of the algorithm which are used to diagonalize against functions that are not exponential honest. We can enumerate all linear-time computable functions with ith function running in time $|i||x| + |i|$. We can then use a universal function for computing $f_i(x)$ in time $2^{\mathcal{O}(|i|+\log(|x|))}$. This will enable Step 8 of our algorithm to be done in $2^{\mathcal{O}(|i|+|x|)}$ steps. All other steps can also be performed in $2^{\mathcal{O}(|i|+|x|)}$ steps. Thus, the set M constructed in the proof would be in NE. Since A is complete under two-way DFA reductions, there would be a two-way DFA reduction f_j from M to A. This reduction would be one-one on $\{j\} \times \mathcal{N}$. This reduction gives us a one–one two-way DFA reduction $f(x) = f_j(\langle j, x\rangle)$ from K to A. Using the observations about one–one two-way DFA reductions, the reduction f will be linear honest and hence invertible. In other words, A is one–one, honest (and, hence, length-increasing), invertible complete. Hence, $K \cong^p A$.

5. Recursively enumerable sets. Finally, we consider RE sets. Although they are less central to complexity theory, they have been more thoroughly studied than subrecursive classes and their structure is well understood. Because of this, it might be hoped that powerful methods developed in recursion theory could be used in understanding the structure of complete RE sets with respect to polynomial reductions. If so, some progress might be expected in this setting. A well-known manuscript of Dowd [Dow82] contains the theorem stating that a set that is m-complete for RE sets is also one–one-complete. Dowd's proof involves a nontrivial application of the recursion theorem, as given in that manuscript. We present here a much simpler proof here which avoids the use of the recursion theorem. Our proof is easily extended to show that the one–one complete set is, in fact, exponentially honest complete.

Let $\Phi_1, \Phi_2, \Phi_3, \ldots$ be an enumeration of all Turing machines. Define

$$K = \{x \mid x \in \Phi_x\} \quad \text{and} \quad K_0 = \{\langle x, y\rangle \mid y \in \Phi_x\}.$$

The following propositions are used later in this section. Their proofs are straightforward.

PROPOSITION 3. *K_0 has a one–one, length-increasing and invertible padding function.*

PROPOSITION 4. *K and K_0 are one–one, length-increasing complete for* RE. *In fact, they are polynomial-time isomorphic.*

THEOREM 7. *If A is a polynomial-time many–one complete set for the class of* RE *sets, then A is polynomial-time one–one complete.*

Proof. Let $f_1, f_2, \ldots$ be any enumeration of all polynomial-time computable functions. Consider the RE set M that is the set of pairs accepted by the following machine.

(1) **input** $\langle i, x\rangle$
(2) **if** $\exists\, y < x$ such that $f_i(\langle i, x\rangle) = f_i(\langle i, y\rangle)$
(3) **then** reject $\langle i, x\rangle$.
(4) **else** accept $\langle i, x\rangle$ if either (a) or (b) holds.
(5) (a) $x \in K$
(6) (b) $\exists\, y > x$ such that $f_i(\langle i, x\rangle) = f_i(\langle i, y\rangle)$

Let f_j be any m-reduction from M to A. Define $f(x) = f_j(\langle j, x\rangle)$. We claim that f is a one–one reduction from K to A. Assume not. Let x_2 be the least element such that for some $x_1 < x_2$, $f_j(\langle j, x_1\rangle) = f_j(\langle j, x_2\rangle)$. By definition of M, $\langle j, x_1\rangle$ would be in M and $\langle j, x_2\rangle$ would be in $\overline{M}$. Therefore, f_j cannot be a reduction from M to A. Contradiction. Hence, f is one–one. Since f is one–one, only Step 5 will be executed on elements of the form $\langle j, x\rangle$. Thus, f is a reduction from K to A.

THEOREM 8. *Let A be any m-complete set for* RE. *Then A is one–one, exponentially honest complete.*

Proof. By Theorem 7, A is one–one complete. Let f_1 be a one–one reduction from K_0 to A. Let $q(\langle i, y\rangle)$ be a function that takes an index i of some machine, a pad y, and returns an index of machine i padded with y. q is one–one, length-increasing and polynomial-time computable. Define f as follows:

$$f(\langle i, x\rangle) = \max\{f_1(\langle q(\langle i, 1\rangle), x\rangle), f_1(\langle q(\langle i, 2\rangle), x\rangle), \cdots f_1(\langle q(\langle i, |\langle i, x\rangle|\rangle), x\rangle)\}.$$

Given two different pairs p_1 and p_2, $f(p_1) \neq f(p_2)$ since they are maximums of two disjoint sets. Thus, f is one–one. Clearly, f is a reduction from K_0 to A. Also, $|f(\langle i, x\rangle)| \geq \log(|\langle i, x\rangle|)$, since $f(\langle i, x\rangle)$ is the maximum of $|\langle i, x\rangle|$ many distinct strings. Hence, f is exponentially honest. This implies that A is one–one, exponentially honest complete.

6. Open problems. We have presented a simple general method of proving that many–one complete sets are one–one complete. However, many interesting open questions remain.

For nondeterministic classes or for RE sets, it is not known how to increase the honesty of reductions to complete sets past exponential honesty. By Proposition 2, if we could show that one–one complete sets for these classes are polynomially honest complete, then they would be length increasing complete. So far, we have been unable to show the polynomially honest completeness, although we believe it is true.

Decreasing the complexity of the classes to which our results apply is also of interest. For deterministic classes, it is not known if our results can be made to apply to subexponential classes that are closed under complement, for example, PSPACE. Of course, of central interest is the same problem for NP-complete sets.

Acknowledgment. We would like to thank Alan Selman for many helpful comments and for a discourse on the naming of complexity classes. We also thank Tim Long and the anonymous referee whose comments have significantly improved the quality of this paper. Our thanks are also due to Eric Allender for his valuable comments on 1–L and two-way reductions. Correspondence with Osamu Watanabe is also very much appreciated.

REFERENCES

[All85] E. W. ALLENDER, *Invertible functions*, Ph.D. thesis, Georgia Institute of Technology, School of Information and Computer Science, Atlanta, GA, 1985.

[All88] ———, *Isomorphisms and* 1-1 *reductions*, J. Comput. System Sci., 36 (1988), pp. 336–350.

[Ber77] L. BERMAN, *Polynomial reducibilities and complete sets*, Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1977.

[BH77] L. BERMAN AND J. HARTMANIS, *On isomorphism and density of* NP *and other complete sets*, SIAM J. Comput., 1 (1977), pp. 305–322.

[Dow82] M. DOWD, *Isomorphism of complete sets*, Tech. Report LCSR-TR-34, Rutgers University, New Brunswick, NJ, 1982.

[HIM78] J. HARTMANIS, N. IMMERMAN, AND S. MAHANEY, *One-way log-tape reductions*, in Proc. 19th IEEE Symposium on the Foundation of Computer Science, Ann Arbor, MI, 1978, pp. 65–72.

[HU79] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[JY85] D. JOSEPH AND P. YOUNG, *Some remarks on witness functions for polynomial reducibilities in* NP, Theoret. Comput. Sci., 39 (1985), pp. 225–237.

[KLD87] K. KO, T. LONG, AND D. DU, *A note on one-way functions and polynomial-time isomorphisms*, Theoret. Comput. Sci., 47 (1987), pp. 263–276.

[KMR88] S. KURTZ, R. MAHANEY, AND S. ROYER, *Collapsing degrees*, J. Comput. System Sci., 37 (1988), pp. 247–268.

[Rog67] H. ROGERS, *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, New York, 1967. Reprinted by MIT Press, Cambridge, MA, 1987.

[Sto74] L. J. STOCKMEYER, *The complexity of decision problems in automata theory and logic*, Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, 1974.

[Wat85] O. WATANABE, *On one-one polynomial time equivalence relations*, Theoret. Comput. Sci., 38 (1985), pp. 157–165.

[You83] P. YOUNG, *Some structural properties of polynomial reducibilities and sets in* NP, in Proc. 15th ACM Symposium on the Theory of Computing, 15 (1983), pp. 392–401.

SIAM J. COMPUT.
Vol. 21, No. 4, pp. 743–754, August 1992

© 1992 Society for Industrial and Applied Mathematics
008

ON THE STRUCTURE OF BOUNDED QUERIES TO ARBITRARY NP SETS*

RICHARD CHANG†

Abstract. Kadin [*SIAM J. Comput.*, 17 (1988), pp. 1263–1282] showed that if the polynomial hierarchy (PH) has infinitely many levels, then for all k, $\mathrm{P}^{\mathrm{SAT}[k]} \subset \mathrm{P}^{\mathrm{SAT}[k+1]}$. This paper extends Kadin's technique and shows that a proper query hierarchy is not an exclusive property of NP complete sets. In fact, for any $A \in \mathrm{NP} - \widehat{\mathrm{low}}_3$, if PH has infinitely many levels, then $\mathrm{P}^{A[k]} \subset \mathrm{P}^{A[k+1]}$. Moreover, for the case of parallel queries, $\mathrm{P}^{A\|[k+1]}$ is not even contained in $\mathrm{P}^{\mathrm{SAT}\|[k]}$. These same techniques can be used to explore some other questions about query hierarchies. For example, if there exists any A such that $\mathrm{P}^{A[2]} = \mathrm{P}^{\mathrm{SAT}[1]}$, then PH collapses to Δ_3^{P}.

Key words. polynomial-time hierarchy, Boolean hierarchy, bounded queries, sparse sets, nonuniform computation

AMS(MOS) subject classifications. 68Q15, 03D15, 03D20

1. Introduction. Soon after the Boolean hierarchy (BH) was defined, the field of computational complexity theory experienced an exciting period when many hierarchies previously believed to be proper were shown to have collapsed [5]. These collapses led many to wonder if BH might also collapse. However, Kadin [6] brought some respect to the Boolean hierarchy by showing that if BH collapses, then so does the polynomial hierarchy (PH). Thus, if the polynomial hierarchy has infinitely many levels, then so would the Boolean hierarchy and the intertwined query hierarchies.

The Boolean hierarchy and the query hierarchy (QH) are built on top of SAT and $\overline{\mathrm{SAT}}$. These hierarchies are contained in the Δ_2^{P} level of the polynomial hierarchy, and their basic properties have been fully explored [1], [2],[6], [13]. BH is defined by combining SAT and $\overline{\mathrm{SAT}}$ with logical "and" and "or." QH is defined by allowing any polynomial time computable combination of SAT and $\overline{\mathrm{SAT}}$.

In this paper we look at hierarchies built on top of arbitrary sets in NP. We want to know when these hierarchies are proper and how they relate to BH and QH. We study these questions in the setting of Schöning's high–low sets and produce results about these hierarchies similar to those already known about BH and QH. In particular, we conclude that if a set is high and the query hierarchy over this set collapses, then PH collapses. We also show that if PH is infinite, then bounded query hierarchies built from many languages in NP are also proper. (By "many," we mean all sets in $\mathrm{NP} - \widehat{\mathrm{low}}_3$.) We interpret this as evidence that a proper QH is not an exclusive property of NP-hard sets. To demonstrate this claim, we show that if PH is infinite, then we can construct a language B in NP so that for all k, $\mathrm{P}^{B[k+1]} \not\subseteq \mathrm{P}^{\mathrm{SAT}[k]}$. Moreover, this language B is not a high set, so it is not hard for NP in any sense (e.g., it is not NP-hard under many–one, Turing, or strong nondeterministic reductions).

2. The Boolean hierarchy and the query hierarchy. In this section, we define the Boolean and Query Hierarchies and summarize some previous work in this area. This is neither a complete nor a chronological summary. We encourage the reader to consult the literature for completeness [1], [2], [3],[6], [13].

We assume that the reader is familiar with P, NP, PH, and oracle Turing machines. We quickly define some familiar notation and concepts. For any language A, we write

*Received by the editors September 8, 1989; accepted for publication July 26, 1991. This research was supported in part by National Science Foundation research grants DCR-8520597 and CCR-88-23053.

†Department of Computer Science, Cornell University, Ithaca, New York 14853.

$A^{=n}$ for the set of strings in A of length n. Recall that a set S is *sparse* if $\|S^{=n}\|$ is bounded by a polynomial. A *padded prefix* of a string x is a string w such that $|w| = |x|$, $w = y\#^k$ and y is a prefix of x. A set S is *self-p-printable* if there is a polynomial time oracle Turing machine D_i^S that prints out $S^{=n}$ on input 1^n. All self-p-printable sets are sparse. A set A is closed under padded prefixes if $w \in A$ whenever $x \in A$ and w is a padded prefix of x. If a set is sparse and closed under padded prefixes, it is self-p-printable.

We now define the parallel query hierarchy ($\mathrm{QH}_{\|}$). This hierarchy is defined by restricting a $\mathrm{P}^{\mathrm{SAT}}$ machine's access to the SAT oracle in two ways. First, the number of queries is limited to a constant (i.e., it does not depend on input length). Second, all the queries must be made at the same time. We allow the P-base machine to do some computation to determine what the queries are, but we do not allow the query strings to depend on the results (oracle answers) of previous queries. We write $\mathrm{P}^{\mathrm{SAT}\|[k]}$ for the class of languages accepted by polynomial time Turing machines that make at most k parallel queries to SAT on input strings of any length. Of course,

$$\mathrm{P}^{\mathrm{SAT}\|[1]} \subseteq \mathrm{P}^{\mathrm{SAT}\|[2]} \subseteq \cdots \subseteq \mathrm{P}^{\mathrm{SAT}\|[k]} \subseteq \mathrm{P}^{\mathrm{SAT}\|[k+1]} \subseteq \cdots.$$

This nested sequence has the upward collapsing property in the sense that if $\mathrm{P}^{\mathrm{SAT}\|[k]} = \mathrm{P}^{\mathrm{SAT}\|[k+1]}$, then the entire parallel query hierarchy ($\mathrm{QH}_{\|} = \bigcup_{j=1}^{\infty} \mathrm{P}^{\mathrm{SAT}\|[j]}$) is contained in $\mathrm{P}^{\mathrm{SAT}\|[k]}$. If PH has infinitely many levels, then $\mathrm{QH}_{\|}$ does, too [6]:

$$\mathrm{P}^{\mathrm{SAT}\|[1]} \subset \mathrm{P}^{\mathrm{SAT}\|[2]} \subset \cdots \subset \mathrm{P}^{\mathrm{SAT}\|[k]} \subset \mathrm{P}^{\mathrm{SAT}\|[k+1]} \subset \cdots.$$

When we remove the parallel query restriction from $\mathrm{P}^{\mathrm{SAT}\|[k]}$ and allow subsequent queries to depend on answers to previous queries, we obtain the serial query hierarchy. We write $\mathrm{P}^{\mathrm{SAT}[k]}$ for the class of languages accepted by polynomial time Turing machines that ask at most k serial queries to SAT for input strings of any length and QH for $\bigcup_{j=1}^{\infty} \mathrm{P}^{\mathrm{SAT}[j]}$. Beigel [1] showed by a clever binary search routine (the mind change technique) that

$$\mathrm{P}^{\mathrm{SAT}\|[2^k-1]} = \mathrm{P}^{\mathrm{SAT}[k]}.$$

Thus, the levels of QH are simply levels of $\mathrm{QH}_{\|}$ that are exponentially far apart. Serial queries are generally considered more natural and relevant than parallel queries, but the finer structure of $\mathrm{QH}_{\|}$ makes it more amenable to analysis.

Next, we examine the Boolean hierarchy [2]. Like PH, each level of BH is composed of two complementary language classes BH_k and co-BH_k, such that

$$\text{co-}\mathrm{BH}_k = \{ L \mid \overline{L} \in \mathrm{BH}_k \}.$$

BH_k is defined inductively, starting with NP and building up with unions and intersections, as follows:

$$\mathrm{BH}_1 \stackrel{\text{def}}{=} \mathrm{NP},$$
$$\mathrm{BH}_{2k} \stackrel{\text{def}}{=} \{ L_1 \cap \overline{L_2} \mid L_1 \in \mathrm{BH}_{2k-1} \text{ and } L_2 \in \mathrm{NP} \},$$
$$\mathrm{BH}_{2k+1} \stackrel{\text{def}}{=} \{ L_1 \cup L_2 \mid L_1 \in \mathrm{BH}_{2k} \text{ and } L_2 \in \mathrm{NP} \}.$$

The levels of BH are interlaced, much like PH:

$$\mathrm{BH}_k \subseteq \mathrm{BH}_{k+1} \cap \text{co-}\mathrm{BH}_{k+1} \subseteq \mathrm{BH}_{k+1},$$
$$\text{co-}\mathrm{BH}_k \subseteq \mathrm{BH}_{k+1} \cap \text{co-}\mathrm{BH}_{k+1} \subseteq \text{co-}\mathrm{BH}_{k+1}.$$

BH also has the upward-collapsing property:

$$\mathrm{BH}_k = \text{co-}\mathrm{BH}_k \Longrightarrow \mathrm{BH} = \mathrm{BH}_k \cap \text{co-}\mathrm{BH}_k.$$

Many results in this area depend on the fact that BH and $\mathrm{QH}_{\|}$ are intertwined [1], [2]:

$$\mathrm{BH}_k \cup \text{co-}\mathrm{BH}_k \subseteq \mathrm{P}^{\mathrm{SAT}\|[k]} \subseteq \mathrm{BH}_{k+1} \cap \text{co-}\mathrm{BH}_{k+1} \subseteq \mathrm{P}^{\mathrm{SAT}\|[k+1]}.$$

Clearly BH is a proper hierarchy if and only if $\mathrm{QH}_{\|}$ is a proper hierarchy. Kadin showed that if BH collapses, then PH collapses. His proof depends on the structure of the canonical complete languages for the levels of BH. For BH_2 and BH_3 these complete languages are:

$$\mathrm{SAT}\wedge\overline{\mathrm{SAT}} \stackrel{\text{def}}{=} \{ \langle F_1, F_2\rangle \mid F_1 \in \mathrm{SAT} \text{ and } F_2 \in \overline{\mathrm{SAT}} \},$$

$$(\mathrm{SAT}\wedge\overline{\mathrm{SAT}})\vee\mathrm{SAT} \stackrel{\text{def}}{=} \{ \langle F_1, F_2, F_3\rangle \mid \langle F_1, F_2\rangle \in \mathrm{SAT}\wedge\overline{\mathrm{SAT}} \text{ or } F_3 \in \mathrm{SAT} \}.$$

3. High and low sets. Schöning [10] defined the high and low hierarchies for NP to classify sets between P and NP. Very roughly, the high-low classification measures the amount of information an NP language (acting as an oracle) can give to a Σ_k^{P} machine. If $A \in \mathrm{NP}$, then for all k

$$\Sigma_k^{\mathrm{P}} \subseteq \Sigma_k^{\mathrm{P},A} \subseteq \Sigma_k^{\mathrm{P},\mathrm{SAT}} = \Sigma_{k+1}^{\mathrm{P}}.$$

If $\Sigma_k^{\mathrm{P},A} = \Sigma_k^{\mathrm{P},\mathrm{SAT}}$, then one might say that the oracle A tells the Σ_k^{P} base machine a lot—as much as SAT does. If $\Sigma_k^{\mathrm{P}} = \Sigma_k^{\mathrm{P},A}$, then one could say that A does not tell the Σ_k^{P} base machine anything that it could not compute itself. In the first case we call A a high set, and in the latter we call it a low set. More formally, we define

$$\mathrm{high}_k \stackrel{\text{def}}{=} \{ A \mid A \in \mathrm{NP} \text{ and } \Sigma_k^{\mathrm{P},A} = \Sigma_k^{\mathrm{P},\mathrm{SAT}} \},$$

$$\mathrm{low}_k \stackrel{\text{def}}{=} \{ A \mid A \in \mathrm{NP} \text{ and } \Sigma_k^{\mathrm{P}} = \Sigma_k^{\mathrm{P},A} \}.$$

Clearly, $\mathrm{low}_k \subseteq \mathrm{low}_{k+1}$ and $\mathrm{high}_k \subseteq \mathrm{high}_{k+1}$. However, the high and low hierarchies are not known to have the familiar upward collapsing behavior. For example, it is not known whether $\mathrm{low}_2 = \mathrm{low}_3$ would imply that $\mathrm{low}_2 = \mathrm{low}_4$.

Schöning also defined a refinement of the low hierarchy [11],

$$\widehat{\mathrm{low}}_k \stackrel{\text{def}}{=} \{ A \mid A \in \mathrm{NP} \text{ and } \Delta_k^{\mathrm{P}} = \Delta_k^{\mathrm{P},A} \}.$$

These classes lie between the levels of the low hierarchy,

$$\mathrm{low}_0 \subseteq \widehat{\mathrm{low}}_1 \subseteq \mathrm{low}_1 \subseteq \widehat{\mathrm{low}}_2 \subseteq \mathrm{low}_2 \subseteq \widehat{\mathrm{low}}_3 \subseteq \mathrm{low}_3$$

and allow a finer classification of language classes in the low hierarchy. In this paper, we will pay special attention to the class $\widehat{\mathrm{low}}_3$.

High and low sets have many interesting properties. Some are briefly mentioned here. Again, we ask the reader to consult the literature for the proofs [7], [9]–[12].

(1) $\mathrm{low}_k = \mathrm{high}_k \iff \mathrm{PH} \subseteq \Sigma_k^{\mathrm{P}}$.

(2) If PH is infinite, then $\exists I \in \mathrm{NP}\ \forall k [I \notin \mathrm{high}_k \text{ and } I \notin \mathrm{low}_k]$.

(3) If S is sparse and $S \in \mathrm{NP}$, then $S \in \widehat{\mathrm{low}}_2$.

(4) If $A \in$ NP and for some sparse set S, $A \in \mathrm{P}^S$, then $A \in \mathrm{low}_3$.
(5) $\mathrm{R} \subseteq \mathrm{BPP} \cap \mathrm{NP} \subseteq \mathrm{low}_2$.
(6) $\mathrm{high}_0 = \{ A \mid A \text{ is } \leq_{\mathrm{T}}^{\mathrm{P}}\text{-complete for NP} \}$.
(7) $\mathrm{low}_0 = \mathrm{P}$.
(8) $\mathrm{low}_1 = \mathrm{NP} \cap \text{co-NP}$.
(9) Graph Isomorphism $\in \mathrm{low}_2$.

4. Main theorem. The three hierarchies QH, $\mathrm{QH}_{\|}$, and BH are built on top of SAT and $\overline{\mathrm{SAT}}$. We now consider analogous query hierarchies built on top of an arbitrary set in NP. For any set $A \in$ NP we can consider the classes $\mathrm{P}^{A\|[k]}$ and $\mathrm{P}^{A[k]}$, defined analogously to $\mathrm{P}^{\mathrm{SAT}\|[k]}$ and $\mathrm{P}^{\mathrm{SAT}[k]}$. We immediately have

$$\mathrm{P}^{A\|[k]} \subseteq \mathrm{P}^{A\|[k+1]}, \quad \mathrm{P}^{A[k]} \subseteq \mathrm{P}^{A[k+1]} \quad \text{and } \mathrm{P}^{A\|[k]} \subseteq \mathrm{P}^{A[k]}.$$

However, in general, we do not know how to repeat Beigel's mind change technique, so we can only relate parallel and serial queries loosely:

$$\mathrm{P}^{A\|[k]} \subseteq \mathrm{P}^{A[k]} \subseteq \mathrm{P}^{A\|[2^k-1]}.$$

The serial query hierarchy over an arbitrary set has the upward-collapsing property, too. That is,

$$\mathrm{P}^{A[k+1]} = \mathrm{P}^{A[k]} \Longrightarrow \forall j > k, \quad \mathrm{P}^{A[j]} = \mathrm{P}^{A[k]}.$$

However, it is not known if the corresponding property holds for the parallel query hierarchy over A. Because of these difficulties, we do not attempt to define a Boolean hierarchy based on $A \in$ NP. Instead, we define *Boolean languages* analogous to the canonical complete languages for BH_k and co-BH_k.

DEFINITION 4.1. For each language A we define a sequence of Boolean languages

$$\mathrm{BL}_1(A) \stackrel{\text{def}}{=} A,$$

$$\mathrm{BL}_{2k}(A) \stackrel{\text{def}}{=} \{ \langle x_1, \cdots, x_{2k} \rangle \mid \langle x_1, \cdots, x_{2k-1} \rangle \in \mathrm{BL}_{2k-1}(A) \text{ and } x_{2k} \in \overline{A} \},$$

$$\mathrm{BL}_{2k+1}(A) \stackrel{\text{def}}{=} \{ \langle x_1, \cdots, x_{2k+1} \rangle \mid \langle x_1, \cdots, x_{2k} \rangle \in \mathrm{BL}_{2k}(A) \text{ or } x_{2k+1} \in A \},$$

$$\text{co-BL}_1(A) \stackrel{\text{def}}{=} \overline{A},$$

$$\text{co-BL}_{2k}(A) \stackrel{\text{def}}{=} \{ \langle x_1, \cdots, x_{2k} \rangle \mid \langle x_1, \cdots, x_{2k-1} \rangle \in \text{co-BL}_{2k-1}(A) \text{ or } x_{2k} \in A \},$$

$$\text{co-BL}_{2k+1}(A) \stackrel{\text{def}}{=} \{ \langle x_1, \cdots, x_{2k+1} \rangle \mid \langle x_1, \cdots, x_{2k} \rangle \in \text{co-BL}_{2k}(A) \text{ and } x_{2k+1} \in \overline{A} \}.$$

Note that $\mathrm{BL}_2(\mathrm{SAT}) = \mathrm{SAT} \wedge \overline{\mathrm{SAT}}$ and $\mathrm{BL}_3(\mathrm{SAT}) = (\mathrm{SAT} \wedge \overline{\mathrm{SAT}}) \vee \mathrm{SAT}$. In the general case, $\mathrm{BL}_k(\mathrm{SAT})$ is $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for BH_k and co-$\mathrm{BL}_k(\mathrm{SAT})$ is $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for co-$\mathrm{BH}_k$. Clearly, $\mathrm{BL}_k(A) \in \mathrm{P}^{A\|[k]}$ and co-$\mathrm{BL}_k(A) \in \mathrm{P}^{A\|[k]}$. Therefore, there is some interaction between the Boolean languages over A and the query hierarchies over A.

With all these definitions in place we can pose some questions that were originally posed for SAT. For example, we would like to know what happens if $\mathrm{P}^{A\|[k]} = \mathrm{P}^{A\|[k+1]}$, if $\mathrm{P}^{A[k]} = \mathrm{P}^{A[k+1]}$, or if $\mathrm{BL}_k(A) \leq_{\mathrm{m}}^{\mathrm{P}}$ co-$\mathrm{BL}_k(A)$. Also, we would like to know the relationship between queries to SAT and queries to A. We know $\mathrm{P}^{A\|[k]} \subseteq \mathrm{P}^{\mathrm{SAT}\|[k]}$, but what is the relationship between $\mathrm{P}^{A\|[k]}$ and $\mathrm{P}^{\mathrm{SAT}\|[k-1]}$? Instead of asking "Is one query to SAT

as powerful as two?" we ask, "Is one query to SAT as powerful as two queries to some other oracle?" The following theorem answers some of these questions.

THEOREM 4.2. *If* $A, B \in \mathrm{NP}$ *and* $\mathrm{BL}_k(A) \leq^{\mathrm{P}}_{\mathrm{m}} \text{co-}\mathrm{BL}_k(B)$, *then* $A \in \widehat{\mathrm{low}}_3$.

We prove this theorem in two parts. Lemma 4.3 is a rather technical lemma and follows the same lines as Kadin's original proof that $\mathrm{BL}_k(\mathrm{SAT}) \leq^{\mathrm{P}}_{\mathrm{m}} \text{co-}\mathrm{BL}_k(\mathrm{SAT})$ implies $\mathrm{PH} \subseteq \Delta^{\mathrm{P}}_3$. We relegate the proof of Lemma 4.3 to the next section.

LEMMA 4.3. *If* $B \in \mathrm{NP}$ *and* $\mathrm{BL}_k(A) \leq^{\mathrm{P}}_{\mathrm{m}} \text{co-}\mathrm{BL}_k(B)$, *then there exists a self-p-printable set* $S \in \Delta^{\mathrm{P}}_3$ *such that* $\overline{A} \in \mathrm{NP}^S$. *(N.B. In Lemma 4.3, we do not assume* $A \in \mathrm{NP}$.*)*

LEMMA 4.4. *If* $A \in \mathrm{NP}$ *and there exists a self-p-printable set* $S \in \Delta^{\mathrm{P}}_3$ *such that* $\overline{A} \in \mathrm{NP}^S$, *then* $A \in \widehat{\mathrm{low}}_3$.

Proof. If $\overline{A} \in \mathrm{NP}^S$, then $\mathrm{NP}^A \subseteq \mathrm{NP}^S$. (To answer a query to A, the NP^S machine runs the NP algorithm for A and the NP^S algorithm for $\overline{A}$ in parallel. One of the algorithms will be correct.) Therefore, we have

$$\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}^{A}}} \subseteq \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}^{S}}}$$

by replacing the NP^A oracle with an NP^S oracle. However, $S \in \Delta^{\mathrm{P}}_3$ and is self-p-printable, so a Δ^{P}_3 machine can write down an initial segment of S that includes all queries to S in a $\Delta^{\mathrm{P},S}_3$ computation. (The length of this initial segment is bounded by a polynomial.) Since Δ^{P}_3 consists of a P-base machine and an $\mathrm{NP}^{\mathrm{NP}}$ oracle, the P-base machine (with the help of the $\mathrm{NP}^{\mathrm{NP}}$ oracle) can write down this initial segment and send it with subsequent oracle queries to $\mathrm{NP}^{\mathrm{NP}}$. With this extra advice, the $\mathrm{NP}^{\mathrm{NP}}$ oracle does not need to consult an S oracle, so

$$\Delta^{\mathrm{P},A}_3 \subseteq \Delta^{\mathrm{P},S}_3 \subseteq \Delta^{\mathrm{P}}_3.$$

Therefore, $A \in \widehat{\mathrm{low}}_3$. □

The theorem gives us a sufficient technical condition for a set to be in $\widehat{\mathrm{low}}_3$. The following corollary clarifies the picture somewhat.

COROLLARY 4.5. *Let* A *be any language in* NP. *If one of the following conditions holds, then* $A \in \widehat{\mathrm{low}}_3$.

(1) $\mathrm{P}^{A[2]} \subseteq \mathrm{P}^{\mathrm{SAT}[1]}$.
(2) $\mathrm{P}^{A\|[k+1]} \subseteq \mathrm{P}^{\mathrm{SAT}\|[k]}$ *for some* $k \geq 1$.
(3) $\mathrm{P}^{A\|[k+1]} = \mathrm{P}^{A\|[k]}$ *for some* $k \geq 1$.
(4) $\mathrm{P}^{A[k+1]} = \mathrm{P}^{A[k]}$ *for some* $k \geq 1$.

Proof. To prove Part 1, assume $\mathrm{P}^{A[2]} \subseteq \mathrm{P}^{\mathrm{SAT}[1]}$. Then, $\mathrm{BL}_2(A) \leq^{\mathrm{P}}_{\mathrm{m}} \text{co-}\mathrm{BL}_2(\mathrm{SAT})$ because $\mathrm{BL}_2(A) \in \mathrm{P}^{A\|[2]}$, $\mathrm{P}^{A\|[2]} \subseteq \mathrm{P}^{A[2]} \subseteq \mathrm{P}^{\mathrm{SAT}[1]} \subseteq \text{co-}\mathrm{BH}_2$, and $\text{co-}\mathrm{BL}_2(\mathrm{SAT})$ is $\leq^{\mathrm{P}}_{\mathrm{m}}$-complete for $\text{co-}\mathrm{BH}_2$. Thus, by the theorem, $A \in \widehat{\mathrm{low}}_3$.

To prove Part 2, assume that $\mathrm{P}^{A\|[k+1]} \subseteq \mathrm{P}^{\mathrm{SAT}\|[k]}$. Then

$$\mathrm{BL}_{k+1}(A) \in \mathrm{P}^{A\|[k+1]} \subseteq \mathrm{P}^{\mathrm{SAT}\|[k]} \subseteq \text{co-}\mathrm{BH}_{k+1}.$$

Since $\text{co-}\mathrm{BL}_{k+1}(\mathrm{SAT})$ is $\leq^{\mathrm{P}}_{\mathrm{m}}$-complete for $\text{co-}\mathrm{BH}_{k+1}$, there exists a many-one reduction from $\mathrm{BL}_{k+1}(A)$ to $\text{co-}\mathrm{BL}_{k+1}(\mathrm{SAT})$. Again, by the theorem, $A \in \widehat{\mathrm{low}}_3$. Part 3 follows from Part 2, since $\mathrm{P}^{A\|[k+1]} = \mathrm{P}^{A\|[k]}$ implies $\mathrm{P}^{A\|[k+1]} \subseteq \mathrm{P}^{\mathrm{SAT}\|[k]}$.

To show Part 4, note that $\mathrm{P}^{A[k+1]} = \mathrm{P}^{A[k]}$ implies $\mathrm{P}^{A[2^k]} \subseteq \mathrm{P}^{A[k]}$ because the whole query hierarchy over A collapses. Thus, $\mathrm{P}^{A\|[2^k]} \subseteq \mathrm{P}^{A[2^k]} \subseteq \mathrm{P}^{A[k]} \subseteq \mathrm{P}^{A\|[2^k-1]}$. Then, $A \in \widehat{\mathrm{low}}_3$ follows from Part 3. □

Parts 3 and 4 of Corollary 4.5 state that if the parallel or serial query hierarchy over A collapses, then A is not very hard. Part 2 says that if a set A is not in $\widehat{\text{low}}_3$, then not only is the parallel query hierarchy over A proper, but it also rises in lock step with $\text{QH}_{\|}$ (see Fig. 1). For NP-hard sets, we can relate Theorem 4.2 to the collapse of PH. In particular, if the query hierarchy over an NP $\leq_{\text{T}}^{\text{P}}$-complete set collapses, then PH collapses to Δ_3^{P}.

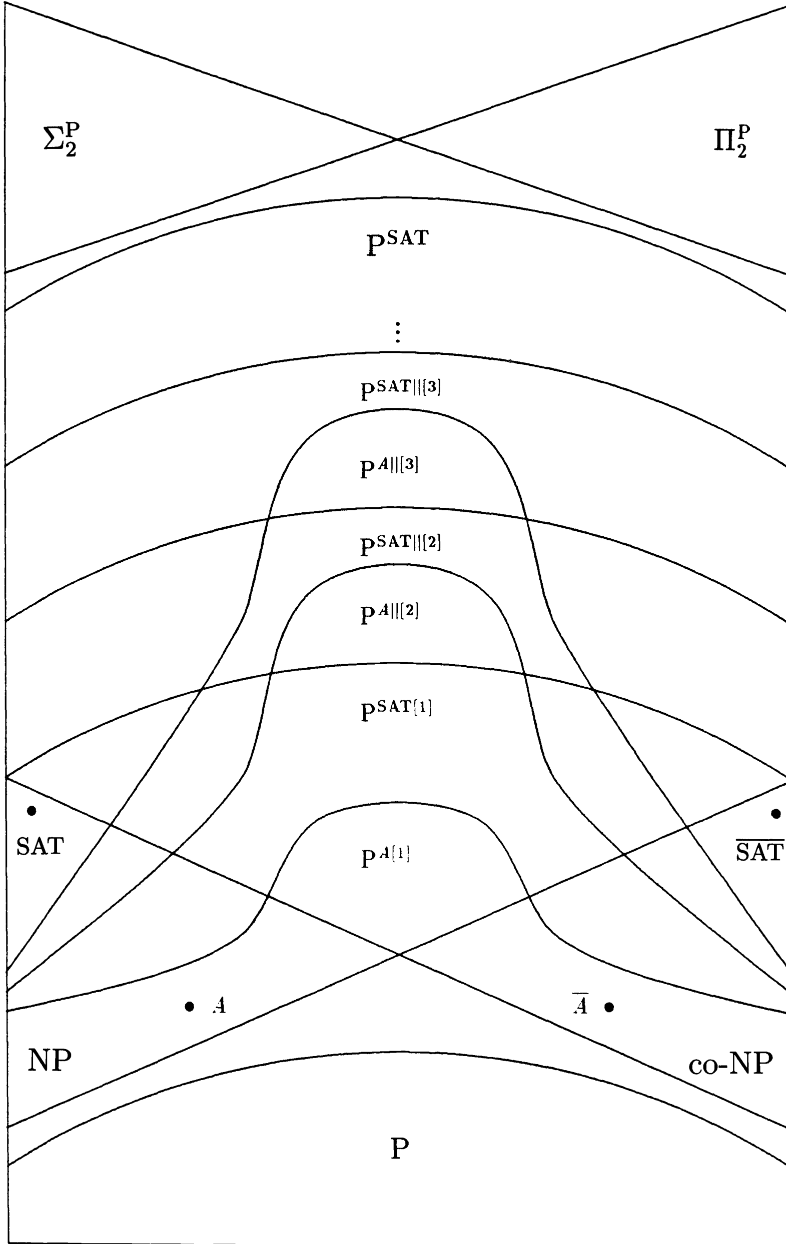


FIG. 1. *Let A be an incomplete set in* NP $-$ $\widehat{\text{low}}_3$.

COROLLARY 4.6. *If* $A \in \text{high}_j$ *for some* j *and one of the following holds, then* PH *is finite.*

(1) $\mathrm{P}^{A[2]} \subseteq \mathrm{P}^{\mathrm{SAT}[1]}$.

(2) $\mathrm{P}^{A\|[k+1]} \subseteq \mathrm{P}^{\mathrm{SAT}\|[k]}$ *for some* $k \geq 1$.

(3) $\mathrm{P}^{A\|[k+1]} = \mathrm{P}^{A\|[k]}$ *for some* $k \geq 1$.

(4) $\mathrm{P}^{A[k+1]} = \mathrm{P}^{A[k]}$ *for some* $k \geq 1$.

Proof. Let $i = \max(3, j)$. By Corollary 4.5, $A \in \widehat{\text{low}_3} \subseteq \text{low}_i$, so $\Sigma_i^{\mathrm{P}} = \Sigma_i^{\mathrm{P},A}$. However, $A \in \text{high}_j \subseteq \text{high}_i$ means $\Sigma_i^{\mathrm{P},A} = \Sigma_{i+1}^{\mathrm{P}}$. Thus, $\Sigma_i^{\mathrm{P}} = \Sigma_i^{\mathrm{P},A} = \Sigma_{i+1}^{\mathrm{P}}$ and $\mathrm{PH} \subseteq \Sigma_i^{\mathrm{P}}$. □

The next corollary generalizes Kadin's result for SAT and answers the question "Is one query to SAT as powerful as two queries to some other oracle?"

COROLLARY 4.7. *If there exists* A *such that* $\mathrm{P}^{A[2]} = \mathrm{P}^{\mathrm{SAT}[1]}$, *then* $\mathrm{PH} \subseteq \Delta_3^{\mathrm{P}}$.

Proof. Suppose that $\mathrm{P}^{A[2]} = \mathrm{P}^{\mathrm{SAT}[1]}$, then $A \in \mathrm{P}^{A[2]} = \mathrm{P}^{\mathrm{SAT}[1]}$. Thus, we know that $A \leq_{\mathrm{m}}^{\mathrm{P}} \mathrm{SAT} \oplus \overline{\mathrm{SAT}}$ via some polynomial time function g, since $\mathrm{SAT} \oplus \overline{\mathrm{SAT}}$ is $\leq_{\mathrm{m}}^{\mathrm{P}}$-complete for $\mathrm{P}^{\mathrm{SAT}[1]}$, where $X \oplus Y$ is defined by

$$X \oplus Y \stackrel{\text{def}}{=} \{\, 0x \mid x \in X \,\} \cup \{\, 1y \mid y \in Y \,\}.$$

We split A into two sets,

$$A_0 = \{\, x \mid x \in A \text{ and } g(x) = 0F \text{ for some } F \,\},$$

$$A_1 = \{\, x \mid x \in A \text{ and } g(x) = 1F \text{ for some } F \,\}.$$

Clearly, $A_0 \in \mathrm{NP}$ and $A_1 \in$ co-NP. Now let $C = A_0 \oplus \overline{A_1}$. One can easily see that $C \in \mathrm{NP}$ and that $\mathrm{P}^{A[k]} = \mathrm{P}^{C[k]}$ for all k. So, $\mathrm{P}^{C[2]} = \mathrm{P}^{A[2]} = \mathrm{P}^{\mathrm{SAT}[1]}$. However, $\mathrm{SAT} \in \mathrm{P}^{C[2]}$ implies C is Turing complete for NP. Thus, C is a high_0 set. Then, by the proof of Corollary 4.6, Part 1, $\mathrm{P}^{C[2]} \subseteq \mathrm{P}^{\mathrm{SAT}[1]}$ implies PH collapses to Δ_3^{P}. □

We can generalize Corollary 4.7 for parallel queries.

COROLLARY 4.8. *If there exists any set* A *such that* $\mathrm{P}^{A\|[r]} = \mathrm{P}^{\mathrm{SAT}\|[k]}$, *and* $r > k > 0$, *then* $\mathrm{PH} \subseteq \Delta_3^{\mathrm{P}}$.

Proof. Since $r > k$, we have $\mathrm{P}^{A\|[k+1]} \subseteq \mathrm{P}^{A\|[r]} = \mathrm{P}^{\mathrm{SAT}\|[k]} \subseteq \text{co-BH}_{k+1}$. Therefore, $\mathrm{BL}_{k+1}(A) \leq_{\mathrm{m}}^{\mathrm{P}} \text{co-BL}_{k+1}(\mathrm{SAT})$. Then by Lemma 4.3 there is a self-p-printable sparse set $S \in \Delta_3^{\mathrm{P}}$ such that $\overline{A} \in \mathrm{NP}^S$. Since, $\mathrm{P}^{\overline{A}\|[k+1]} = \mathrm{P}^{A\|[k+1]} \subseteq \text{co-BH}_{k+1}$, repeating the argument for $\overline{A}$ yields S' such that $A \in \mathrm{NP}^{S'}$. Thus,

$$\mathrm{NP}^A \subseteq \mathrm{NP}^{S \oplus S'}.$$

However, $\mathrm{P}^{A\|[r]} = \mathrm{P}^{\mathrm{SAT}\|[k]}$ implies $\mathrm{SAT} \in \mathrm{P}^A$, so, $\mathrm{NP}^{\mathrm{SAT}} \subseteq \mathrm{NP}^A$. Thus,

$$\mathrm{P}^{\mathrm{NP}^{\mathrm{NP}^{\mathrm{SAT}}}} \subseteq \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}^A}} \subseteq \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}^{S \oplus S'}}} \subseteq \mathrm{P}^{\mathrm{NP}^{\mathrm{NP}}}.$$

Therefore, $\mathrm{PH} \subseteq \Delta_3^{\mathrm{P}}$. □

Corollaries 4.7 and 4.8 say that if PH is infinite, then there is no way to split queries to SAT into exact portions. For example, there would not be a set A where 3 parallel queries to A is exactly 2 parallel queries to SAT. However, we still do not know if there can be sets B such that $\mathrm{P}^{B\|[2]} = \mathrm{P}^{\mathrm{SAT}\|[3]}$. If we can show that such sets do not exist (under certain hypotheses), then we can put forward the thesis that parallel queries to SAT are "atomic" or "indivisible" in some sense.

Now we return to queries to sets in NP. In the next two corollaries, we explore technical conditions that allow us to strengthen the results about the serial query hierarchies.

COROLLARY 4.9. *If* $A \in \mathrm{NP} - \widehat{\mathrm{low}}_3$, *then there exist* $B \in \mathrm{NP}$ *such that* $A \leq_{\mathrm{m}}^{\mathrm{P}} B$, $\mathrm{P}^A = \mathrm{P}^B$ *and for all k*, $\mathrm{P}^{B[k+1]} \not\subseteq \mathrm{P}^{\mathrm{SAT}[k]}$.

Proof. Note that if $\mathrm{P}^B = \mathrm{P}^A$ then A and B are in the same high or low level. So, this corollary also says that for every high and low level above $\widehat{\mathrm{low}}_3$, there is a set B whose *serial* query hierarchy rises in lock step with the *serial* query hierarchy for SAT.

To prove this lemma, we need to mimic Beigel's mind change technique for sets other than SAT, which has some special properties that make the technique work.

DEFINITION 4.10.

$$\mathrm{OR}_2(C) = \{ \langle x, y \rangle \mid x \in C \text{ or } y \in C \},$$

$$\mathrm{AND}_2(C) = \{ \langle x, y \rangle \mid x \in C \text{ and } y \in C \},$$

(where "2" in the subscript means binary).

SAT is a special set because $\mathrm{OR}_2(\mathrm{SAT}) \leq_{\mathrm{m}}^{\mathrm{P}} \mathrm{SAT}$ and $\mathrm{AND}_2(\mathrm{SAT}) \leq_{\mathrm{m}}^{\mathrm{P}} \mathrm{SAT}$. By going over Beigel's mind change proof [1], [4], one can show that if a set $C \in \mathrm{NP}$ has the property that $\mathrm{OR}_2(C) \leq_{\mathrm{m}}^{\mathrm{P}} C$ and $\mathrm{AND}_2(C) \leq_{\mathrm{m}}^{\mathrm{P}} C$, then $\mathrm{P}^{C\|[2^k-1]} = \mathrm{P}^{C[k]}$.

Now we modify A slightly, so that it has the desired properties. Let B be the set of tuples of the form $\langle F, x_1, \cdots, x_n \rangle$ such that F is a Boolean formula over n variables $y_1, \cdots, y_n$ without negation, and when y_i is evaluated as "$x_i \in A$", F evaluates to 1. For example, if $a_1, a_2 \in A$ and $a_3, a_4 \notin A$, then:

$$\langle y_1 \wedge y_2, a_1, a_2 \rangle \in B, \qquad \langle y_1 \wedge y_2, a_1, a_3 \rangle \notin B,$$

$$\langle y_1 \vee y_2, a_1, a_3 \rangle \in B, \qquad \langle y_1 \vee y_2, a_4, a_3 \rangle \notin B.$$

Clearly, B is in NP and $A \leq_{\mathrm{m}}^{\mathrm{P}} B$. Also, $B \in \mathrm{NP} - \widehat{\mathrm{low}}_3$, because $\mathrm{P}^B = \mathrm{P}^A$. Moreover, $\mathrm{OR}_2(B) \leq_{\mathrm{m}}^{\mathrm{P}} B$ and $\mathrm{AND}_2(B) \leq_{\mathrm{m}}^{\mathrm{P}} B$, so $\mathrm{P}^{B\|[2^k-1]} = \mathrm{P}^{B[k]}$. Now if $\mathrm{P}^{B[k+1]} \subseteq \mathrm{P}^{\mathrm{SAT}[k]}$, we know that

$$\mathrm{BL}_{2^k}(B) \in \mathrm{P}^{B\|[2^k]} \subseteq \mathrm{P}^{B\|[2^{k+1}-1]} = \mathrm{P}^{B[k+1]} \subseteq \mathrm{P}^{\mathrm{SAT}[k]} = \mathrm{P}^{\mathrm{SAT}\|[2^k-1]} \subseteq \text{co-BH}_{2^k}.$$

Thus, $\mathrm{BL}_{2^k}(B) \leq_{\mathrm{m}}^{\mathrm{P}} \text{co-BL}_{2^k}(\mathrm{SAT})$ which contradicts the assumption that $B \notin \widehat{\mathrm{low}}_3$. □

We can say more about the existence of incomplete sets whose serial query hierarchies rise in lock step with QH. (Note that it may be the case that none of the sets in $\mathrm{NP} - \widehat{\mathrm{low}}_3$ are incomplete. They may all be Turing complete for NP or complete for NP in some other sense. However, in this case PH is finite.)

COROLLARY 4.11. *If the polynomial hierarchy is infinite, then there exists a set* $B \in \mathrm{NP}$ *such that for all k*, $\mathrm{P}^{B[k+1]} \not\subseteq \mathrm{P}^{\mathrm{SAT}[k]}$, *but B is not high.*

Proof. If PH is infinite, then by a Ladner-like delayed diagonalization [8], [9] we can construct a set $I \in \mathrm{NP}$ that is neither high nor low (I stands for intermediate). In particular, $I \notin \widehat{\mathrm{low}}_3$, so using Corollary 4.9 we can obtain a set B such that for all k, $\mathrm{P}^{B[k+1]} \not\subseteq \mathrm{P}^{\mathrm{SAT}[k]}$. Since $\mathrm{P}^B = \mathrm{P}^I$, B is intermediate if and only if I is intermediate. Therefore, B is not high. □

Note that if B is not high, then B is not NP-hard under many-one, Turing, strong nondeterministic or other assorted reductions. In particular B is not Turing complete for NP so $\mathrm{SAT} \notin \mathrm{P}^B$. So, Corollary 4.11 says that the serial query hierarchy over B rises in lock step with QH but never captures SAT.

5. Proof of Lemma 4.3.

LEMMA 4.3. *If* $B \in \mathrm{NP}$ *and* $\mathrm{BL}_k(A) \leq_m^P \text{co-BL}_k(B)$*, then there exists a self-p-printable set* $S \in \Delta_3^P$ *such that* $\overline{A} \in \mathrm{NP}^S$.

Proof First, note that we do not assume $A \in \mathrm{NP}$. However, the assumption that $\mathrm{BL}_k(A) \leq_m^P \text{co-BL}_k(B)$ implies that $A \in \mathrm{P}^{\mathrm{NP}}$, since

$$A \leq_m^P \mathrm{BL}_k(A), \quad \mathrm{BL}_k(A) \leq_m^P \text{co-BL}_k(B) \quad \text{and} \quad \text{co-BL}_k(B) \in \mathrm{P}^{\mathrm{NP}}.$$

Therefore, by the closure of P^{NP} under $\leq_m^P$ reductions, $A \in \mathrm{P}^{\mathrm{NP}}$.

Now we prove this lemma by producing a Δ_3^P program that on input 1^n generates a finite set T_n with $\leq kn$ elements. Furthermore, every string in T_n will have length n. The set produced, $S = \bigcup_{n\geq 1}^{\infty} T_n$, will have the specified properties. The main part of the program is a loop that is iterated for values of i from 0 up to $k-1$. Each iteration produces either the desired T_n or a "reduction" from $\mathrm{BL}_{k-i}(A)$ to $\text{co-BL}_{k-i}(B)$ for strings of length n.

In the following discussion, we let g be the polynomial time function that reduces $\mathrm{BL}_k(A)$ to $\text{co-BL}_k(B)$ and let $j = k - i$. We also use the following notational devices:

DEFINITION 5.1. Let $\langle x_1, \cdots, x_k\rangle$ be any k-tuple. When the individual strings in the tuple are not significant, we will substitute $\vec{x}$ for $\langle x_1, \cdots, x_k\rangle$. Also, we write $\vec{x}^R$ for $\langle x_k, \cdots, x_1\rangle$, the reversal of the tuple. Finally, we will use $\{0,1\}^{m\times k}$ to denote the set of k-tuples of strings of length m.

DEFINITION 5.2. We will write π_j for the jth projection function, and $\pi_{(i,j)}$ for the function that selects the ith through jth elements of a k-tuple. For example,

$$\pi_j(x_1, \cdots, x_k) = x_j, \quad \text{and} \quad \pi_{(i,j)}(x_1, \cdots, x_k) = \langle x_i, \cdots, x_j\rangle.$$

We maintain the following loop invariant to assist our proof. Before each iteration, we have $\vec{z} = \langle z_1, \cdots, z_i\rangle \in \{0,1\}^{n\times i}$ such that for all $\vec{x} = \langle x_1, \cdots, x_j\rangle \in \{0,1\}^{n\times j}$,

$$\vec{x} \in \mathrm{BL}_j(A) \iff \pi_{(1,j)} \circ g(\vec{x}, \vec{z}^R) \in \text{co-BL}_j(B).$$

This loop invariant holds trivially for $i = 0$, since g is a reduction from $\mathrm{BL}_k(A)$ to $\text{co-BL}_k(B)$. The body of the loop is given in Fig. 2. Note that the loop terminates either at Step 1 or at Step 5.

Now we show how to construct T_n. There are two cases. If $\overline{A}^{=n} = \emptyset$, then we put $@^n$ in T_n. Note that we can easily check if $\overline{A}^{=n} = \emptyset$ with a Π_2^P question, because $A \in \mathrm{P}^{\mathrm{NP}}$. If $\overline{A}^{=n} \neq \emptyset$, then we start the loop described in Fig. 2 with $i = 0$. When the loop terminates, we put $z_1, \cdots, z_i$ and all their padded prefixes in T_n. We still have to prove two claims. First, we must show that the loop invariant holds from iteration to iteration. Also, we need to show that $\overline{A}^{=n} \in \mathrm{NP}^{T_n}$.

CLAIM 1. Suppose that in some iteration i we reach Step 5. From the loop invariant of the ith loop iteration, we know that $\forall \vec{x} = \langle x_1, \cdots, x_j\rangle \in \{0,1\}^{n\times j}$ and

$$\vec{x} \in \mathrm{BL}_j(A) \iff \pi_{(1,j)} \circ g(\vec{x}, \vec{z}^R) \in \text{co-BL}_j(B).$$

Let $\vec{u} = \langle u_1, \cdots, u_j\rangle = \pi_{(1,j)} \circ g(\vec{x}, \vec{z}^R)$. Then,

$$\vec{x} \in \mathrm{BL}_j(A) \iff \vec{u} \in \text{co-BL}_j(B).$$

Now, let $\vec{x}\,' = \langle x_1, \cdots, x_{j-1}\rangle$ and $\vec{u}\,' = \langle u_1, \cdots, u_{j-1}\rangle$. If j is even, then by the definition of $\mathrm{BL}_j(A)$ and $\text{co-BL}_j(B)$ we have

$$\vec{x}\,' \in \mathrm{BL}_{j-1}(A) \quad \text{and} \quad x_j \in \overline{A} \iff \vec{u}\,' \in \text{co-BL}_{j-1}(B) \quad \text{or} \quad u_j \in B.$$

(1) if $i = k-1$, then write down $\vec{z} = \langle z_1, \cdots, z_i \rangle$ and exit the loop.
(2) Compile a function $h : \{0,1\}^{n \times j} \to \{0,1\}^*$ defined by $h(\vec{v}) = \pi_j \circ g(\vec{v}, \vec{z}^R)$.
(3) Ask the $\mathrm{NP}^{\mathrm{NP}}$ oracle,

$$\text{“}\forall x \in \overline{A}^{=n},\ \exists \vec{x}\,' = \langle x_1, \cdots, x_{j-1} \rangle \in \{0,1\}^{n \times (j-1)},\ h(\vec{x}\,', x) \in B?\text{”}$$

(It is important to note here that the question above can be answered by an $\mathrm{NP}^{\mathrm{NP}}$ oracle because $A \in \mathrm{P}^{\mathrm{NP}}$.)
(4) If the oracle answers yes, then write down $\vec{z} = \langle z_1, \cdots, z_i \rangle$ and exit the loop.
(5) If the oracle answers no, then there exists an $x \in \overline{A}^{=n}$ such that

$$\forall \vec{x}\,' \in \{0,1\}^{n \times (j-1)},\ h(\vec{x}\,', x) \notin B.$$

By using binary search and the $\mathrm{NP}^{\mathrm{NP}}$ oracle, find the lexically smallest such x and write it down. Finally, let $z_{i+1} := x$, $\vec{z} := \langle z_1, \cdots, z_{i+1} \rangle$, $i := i+1$ and advance to the next iteration.

FIG. 2. *The body of the loop.*

Now, fix x_j to be the lexically smallest x found in Step 5. We know that $x \in \overline{A}$ and $u_j = h(\vec{x}\,', x) \notin B$, so we are left with

$$\vec{x}\,' \in \mathrm{BL}_{j-1}(A) \iff \vec{u}\,' \in \text{co-}\mathrm{BL}_{j-1}(B).$$

Since $\vec{u}\,' = \pi_{(1,j-1)} \circ g(\vec{x}\,', x, z_i, \cdots, z_1)$, this is exactly the loop invariant for the $(i+1)$th iteration when we define z_{i+1} to be x.

In the other case, if j is odd, we have

$$\vec{x}\,' \in \mathrm{BL}_{j-1}(A) \quad \text{or} \quad x_j \in A \iff \vec{u}\,' \in \text{co-}\mathrm{BL}_{j-1}(B) \quad \text{and} \quad u_j \in \overline{B},$$

or (by negating both sides of the if and only if)

$$\vec{x}\,' \notin \mathrm{BL}_{j-1}(A) \quad \text{and} \quad x_j \in \overline{A} \iff \vec{u}\,' \notin \text{co-}\mathrm{BL}_{j-1}(B) \quad \text{or} \quad u_j \in B.$$

Fixing x_j to be the minimal x found in Step 5, we have

$$\vec{x}\,' \notin \mathrm{BL}_{j-1}(A) \iff \vec{u}\,' \notin \text{co-}\mathrm{BL}_{j-1}(B)$$

or (by negating both sides of the if and only if)

$$\vec{x}\,' \in \mathrm{BL}_{j-1}(A) \iff \vec{u}\,' \in \text{co-}\mathrm{BL}_{j-1}(B).$$

Again, this is the loop invariant for the next iteration. Thus, in both cases we manage to maintain the loop invariant.

CLAIM 2. We need to show that $\overline{A}^{=n} \in \mathrm{NP}^{T_n}$. There are two cases to consider.

Case 1. If the loop terminated with $i = k-1$, then $\vec{z} = \langle z_1, \cdots, z_{k-1} \rangle$ and from the loop invariant we know that for all $x \in \{0,1\}^n$ and

$$x \in \mathrm{BL}_1(A) \iff \pi_{(1,1)} \circ g(x, \vec{z}^R) \in \text{co-}\mathrm{BL}_1(B).$$

However, $\mathrm{BL}_1(A) =$ A and $\text{co-}\mathrm{BL}_1(B) = \overline{B}$, so we really have

$$x \in A \iff \pi_1 \circ g(x, \vec{z}^R) \in \overline{B}$$

or (by negating both sides of the if and only if)

$$x \in \overline{A} \iff \pi_1 \circ g(x, \vec{z}^R) \in B.$$

To check if $x \in \overline{A}^{=n}$ an NP^{T_n} machine simply queries T_n to find $\vec{z}$, computes $u = \pi_1 \circ g(x, \vec{z})$ and accepts if and only if $u \in B$.

Case 2. If the loop terminated with $i < k - 1$, then $\vec{z} = \langle z_1, \cdots, z_i \rangle$. We want to show that

$$x \in \overline{A}^{=n} \iff \exists \vec{x}\,' = \langle x_1, \cdots, x_{j-1} \rangle \in \{0,1\}^{n \times (j-1)},\ h(\vec{x}\,', x) \in B,$$

where $j = k - i$ and $h(\vec{v}) = \pi_j \circ g(\vec{v}, \vec{z}^R)$. Since the loop terminated in Step 4, the $\mathrm{NP}^{\mathrm{NP}}$ oracle must have answered yes to the question:

$$\text{“}\forall x \in \overline{A}^{=n}\ \exists \vec{x}\,' = \langle x_1, \cdots, x_{j-1} \rangle \in \{0,1\}^{n \times (j-1)},\ h(\vec{x}\,', x) \in B?\text{”}$$

Therefore, we obtain one direction of the if and only if

$$(1) \qquad x \in \overline{A}^{=n} \Longrightarrow \exists \vec{x}\,' = \langle x_1, \cdots, x_{j-1} \rangle \in \{0,1\}^{n \times (j-1)},\ h(\vec{x}\,', x) \in B,$$

Moreover, we know from the loop invariant that $\forall \vec{x}\,' = \langle x_1, \cdots, x_{j-1} \rangle \in \{0,1\}^{n \times (j-1)}$,

$$\langle \vec{x}\,', x \rangle \in \mathrm{BL}_j(A) \iff \pi_{(1,j)} \circ g(\vec{x}\,', x, \vec{z}^R) \in \text{co-}\mathrm{BL}_j(B).$$

Again, let $\vec{u} = \langle u_1, \cdots, u_j \rangle = \pi_{(1,j)} \circ g(\vec{x}\,', x, \vec{z}^R)$ and let $\vec{u}\,' = \langle u_1, \cdots, u_{j-1} \rangle$. If j is even, then we know from the definition of $\mathrm{BL}_j(A)$ and co-$\mathrm{BL}_j(B)$ that

$$\vec{x}\,' \in \mathrm{BL}_{j-1}(A) \quad \text{and} \quad x \in \overline{A} \iff \vec{u}\,' \in \text{co-}\mathrm{BL}_{j-1}(B) \quad \text{or} \quad u_j \in B.$$

If j is odd, we get

$$\vec{x}\,' \in \mathrm{BL}_{j-1}(A) \quad \text{or} \quad x \in A \iff \vec{u}\,' \in \text{co-}\mathrm{BL}_{j-1}(B) \quad \text{and} \quad u_j \in \overline{B}.$$

Note that $u_j = h(\vec{x}\,', x)$ and that in either case $u_j \in B$ implies $x \in \overline{A}$. Therefore, we obtain the other direction of the if and only if

$$(2) \qquad \exists \vec{x}\,' = \langle x_1, \cdots, x_{j-1} \rangle \in \{0,1\}^{n \times (j-1)},\ h(\vec{x}\,', x) \in B \Longrightarrow x \in \overline{A}.$$

Combining the implications in (1) and (2), we have

$$x \in \overline{A}^{=n} \iff \exists \vec{x}\,' = \langle x_1, \cdots, x_{j-1} \rangle \in \{0,1\}^{n \times (j-1)},\ h(\vec{x}\,', x) \in B,$$

This relationship allows us to compute $\overline{A}^{=n}$ with an NP^{T_n} machine. To check if $x \in \overline{A}^{=n}$, an NP^{T_n} machine queries T_n to find $\vec{z} = \langle z_1, \cdots, z_i \rangle$, guesses $\vec{x}\,' = \langle x_1, \cdots, x_{j-1} \rangle$ and accepts if and only if $\pi_j \circ g(\vec{x}\,', x, \vec{z}^R) \in B$.

In summary, we constructed $S \in \Delta_3^{\mathrm{P}}$ length by length (i.e., $S = \bigcup_{n \geq 1}^{\infty} T_n$). Each T_n has at most kn strings, and all the strings in T_n are of length n. Also, each T_n is closed under prefixes, so S is self-p-printable. Finally, $\overline{A} \in \mathrm{NP}^S$ because the following NP^S program determines if $x \in \overline{A}$:

(1) Let $|x| = n$.
(2) If $@^n \in S$, then $\overline{A}^{=n} = \emptyset$. Reject x.
(3) Print out the strings $z_1, \cdots, z_i$ in $S^{=n}$. Let $j = k - i$.
(4) If $i = k - 1$, compute $u = \pi_1 \circ g(x, \vec{z}^R)$ and accept if and only if $u \in B$.
(5) If $i < k-1$, accept x if and only if $\exists \vec{x}\,' \in \{0,1\}^{n \times (j-1)}, \pi_j \circ g(x_1, \cdots, x_{'}, x, \vec{z}^R) \in B$. □

6. Conclusion. In this paper we have shown that except for the sets in $\widehat{\mathrm{low}}_3$, sets in NP yield proper query hierarchies. In fact, assuming that PH is infinite, there even exist incomplete sets that produce proper query hierarchies.

Many questions remain. For example, we know that query hierarchies over sets in P always collapse. We also know that for any $A \in \mathrm{NP} \cap \text{co-NP}$ there exists $B \in \mathrm{NP} \cap \text{co-NP}$ such that $A \leq_m^P B$ and the query hierarchy over B collapses. Are there sets in $\mathrm{NP} \cap \text{co-NP}$ that have proper query hierarchies? What about sets between $\widehat{\mathrm{low}}_3$ and $\mathrm{NP} \cap \text{co-NP}$? Are their query hierarchies proper? Many interesting sub-classes of NP are contained in this region, including sparse sets in NP, $\mathrm{NP} \cap \mathrm{P/poly}$, R and $\mathrm{BPP} \cap \mathrm{NP}$. Can we show that any of these sets have proper or collapsing hierarchies? Also, we would like to strengthen the results for serial query hierarchies. We know that $\mathrm{AND}_2(A) \leq_m^P A$ and $\mathrm{OR}_2(A) \leq_m^P A$ is a sufficient condition. We know that this condition holds for all $\leq_m^P$-complete sets, all sets in P, and Graph Isomorphism. Primes remains the only candidate for a natural language that does not have this property.

Acknowledgments. The author would like to thank his advisor, Juris Hartmanis, for guidance and support, Jim Kadin for invaluable discussions and for suggesting nice open questions, Wei Li and Desh Ranjan for keeping the author on his toes, Georges Lauri for proofreading the paper, and Christine Piatko for lending him a red pen during a time of great need.

REFERENCES

[1] R. BEIGEL, *Bounded queries to* SAT *and the Boolean hierarchy*, Theoret. Comput. Sci., 84 (1991), pp. 199–223.

[2] J. CAI, T. GUNDERMANN, J. HARTMANIS, L. HEMACHANDRA, V. SEWELSON, K. WAGNER, AND G. WECHSUNG, *The Boolean hierarchy I: Structural properties*, SIAM J. Comput., 17 (1988), pp. 1232–1252.

[3] R. CHANG AND J. KADIN, *The Boolean hierarchy and the polynomial hierarchy: a closer connection*, in Proceedings of the 5th Structure in Complexity Theory Conference, Computer Society Press of the IEEE, July 1990, pp. 169–178.

[4] ———, *On computing Boolean connectives of characteristic functions*, Tech. Report TR 90-1118, Cornell Department of Computer Science, May 1990; *Math. Systems Theory*, to appear.

[5] J. HARTMANIS, *Collapsing hierarchies*, Bull. European Association Theoret. Comput. Sci., 33 (1987), pp. 26–39.

[6] J. KADIN, *The polynomial time hierarchy collapses if the Boolean hierarchy collapses*, SIAM J. Comput., 17 (1988), pp. 1263–1282.

[7] K. KO AND U. SCHÖNING, *On circuit size complexity and the low hierarchy for* NP, SIAM J. Comput., 14 (1985), pp. 41–51.

[8] R. LADNER, *On the structure of polynomial time reducibility*, J. Assoc. Comput. Mach., 22 (1975), pp. 155–171.

[9] U. SCHÖNING, *A uniform approach to obtain diagonal sets in complexity classes*, Theoret. Comput. Sci., 18 (1982), pp. 95–103.

[10] ———, *A low and a high hierarchy in* NP, J. Comput. System Sci., 27 (1983), pp. 14–28.

[11] ———, *Complexity and Structure*, Lecture Notes in Computer Science, 211, Springer-Verlag, Berlin, New York, 1985.

[12] ———, *Graph isomorphism is in the low hierarchy*, in Proc. 4th Sympos. Theoret. Aspects of Comput. Sci., 247, in Lecture Notes in Computer Science, Springer-Verlag, Berlin, New York, 1987, pp. 114–124.

[13] K. WAGNER, *Bounded query computations*, in Proceedings of the 3rd Structure in Complexity Theory Conference, Computer Society Press of the IEEE, June 1988, pp. 260–277.

SIAM J. COMPUT.
Vol. 21, No. 4, pp. 755–780, August 1992

© 1992 Society for Industrial and Applied Mathematics
009

AN OPTIMAL PARALLEL ALGORITHM FOR FORMULA EVALUATION*

S. BUSS†, S. COOK‡, A. GUPTA§, AND V. RAMACHANDRAN¶

Abstract. A new approach to Buss's $\mathbf{NC}^1$ algorithm [*Proc. 19th ACM Symposium on Theory of Computing*, Association for Computing Machinery, New York, 1987, pp. 123–131] for evaluation of Boolean formulas is presented. This problem is shown to be complete for $\mathbf{NC}^1$ over $\mathbf{AC}^0$ reductions. This approach is then used to solve the more general problem of evaluating arithmetic formulas by using arithmetic circuits.

Key words. formula evaluation, Boolean formulas, parallel algorithms, circuit complexity, log-time hierarchy

AMS(MOS) subject classification. 68Q

1. Introduction. In this paper we consider the parallel complexity of the Boolean and arithmetic formula-value problems. The Boolean formula-value problem is that of determining the truth value of an infix Boolean formula (with connectives $\{\wedge, \vee, \neg\}$) given the truth assignments of the variables in the formula. Since it is easy to substitute values for the variables, we can reduce this problem to that of solving the Boolean sentence-value problem (BSVP), i.e., the Boolean formula-value problem restricted to the case in which the formula contains constants and operators, but no variables. The goal is to obtain a bounded fan-in Boolean circuit of small depth that solves the BSVP for all inputs of a given size. We assume that each gate takes unit time for its computation and that there is no propagation delay along wires. This is the standard circuit model (see, e.g., [sa76], [co85], [kr90]). In this model the time taken by a circuit to compute the values of its outputs when given values to its inputs is equal to the depth of the circuit. Hence, a circuit of small depth corresponds to a computation that can be performed quickly in parallel.

A natural extension to the Boolean formula-value problem is the problem of evaluating an arithmetic formula over a more general algebra. In this paper we consider this problem over semi-rings, rings, and fields. The problem is basically the same as BSVP—given an arithmetic formula over an algebra and the value of each variable in the formula, determine the value of the formula. We use the arithmetic-Boolean circuits of von zur Gathen [jg86] as our model, and we use the corresponding arithmetic complexity theory. Once again, the goal is to obtain a circuit of small depth that solves this problem for all inputs of a given depth. We assume that each arithmetic gate has unit delay, so that the time required by the circuit to perform a computation is equal to its depth.

An additional property that we desire in the family of circuits we construct is that it be uniform, i.e., that a description of the circuit for evaluating formulas of size n can be obtained easily when the value of n is known; the family is logspace uniform if the

*Received by the editors April 10, 1989; accepted for publication (in revised form) July 3, 1990.

†Department of Mathematics, University of California, San Diego, La Jolla, California 92093. This research was partially supported by National Science Foundation grants DMS85-11465 and DMS87-01828.

‡Department of Computer Science, University of Toronto, Toronto, Canada, M5S 1A4. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

§Department of Computer Science, University of Toronto, Toronto, Canada, M5S 1A4. Present address, School of Computing Science, Simon Fraser University, Burnaby, British Columbia, Canada, V5A 1S6. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

¶Department of Computer Science, University of Texas at Austin, Austin, Texas 78712. This work was supported in part by National Science Foundation grant ECS 8404866, Semiconductor Research Corporation grant RSCH 84-06-049-6, and Joint Services Electronics Program grant N00014-84-C-0149 while the author was with the University of Illinois, Urbana.

description of the nth circuit can be provided by a deterministic Turing machine operating in space $O(\log n)$. The class $\mathbf{NC}^k$ for $k \geq 2$ is the class of problems that have a logspace-uniform family of circuits of depth $O(\log^k n)$ and polynomial size, where n is the size of the input; for $\mathbf{NC}^1$ a stronger notion of uniformity is usually used [ru81]. The class $\mathbf{NC}$ is the class of problems that have a logspace-uniform family of circuits of polylog depth and polynomial size; this class is generally considered to characterize the class of problems with feasible parallel algorithms. Let $\mathbf{P}$ be the class of problems solvable in sequential polynomial time. An important open question in parallel complexity theory is whether $\mathbf{NC}$ equals $\mathbf{P}$ or whether $\mathbf{NC}^1$ equals $\mathbf{P}$. For more on parallel circuit complexity see, e.g., [co85], [kr90], [ru81].

Simple fan-in arguments show that any circuit for formula evaluation must have depth at least logarithmic in the size of the formula. Early work on BSVP was done by Spira [sp71], who showed that any sentence of size n can be restructured into a formula of depth $O(\log n)$ and size $O(n^2)$. Brent [br74] used a restructured circuit of logarithmic depth and linear size to evaluate a given arithmetic formula. These results gave hope of obtaining a logarithmic-depth circuit for formula evaluation by finding a logarithmic-depth circuit for performing the appropriate restructuring. However, direct implementation of these algorithms seems to require $\Omega(\log^2 n)$ depth for the restructuring. This result placed BSVP in $\mathbf{NC}^2$.

The BSVP can be shown to be in $\mathbf{NC}^2$ through the use of other techniques. Lynch [ly77] showed that parenthesised context-free languages can be recognized in deterministic log space (LOGSPACE). Since the set of true Boolean sentences is an instance of these languages, this immediately implied the same space bound for BSVP. The result of Borodin [bo77], that LOGSPACE $\subseteq \mathbf{NC}^2$, once again placed this problem in $\mathbf{NC}^2$. The logarithmic-time tree-contraction algorithm of Miller and Reif [mr85] for arithmetic expression evaluation on a PRAM again translates into an $\mathbf{NC}^2$ algorithm on arithmetic circuits.

The first sub-$\mathbf{NC}^2$ algorithm for BSVP was devised by Cook and Gupta [gu85] and independently by Ramachandran [ra86]. Their circuit family for the problem was log space uniform and had a depth of $O(\log n \log\log n)$, and this gave new hope that the problem had an $\mathbf{NC}^1$ algorithm. Cook and Gupta also showed that parenthesis context-free grammars can be recognized in depth $O(\log n \log\log n)$, while Ramachandran showed that arithmetic formulas over semi-rings can be evaluated within the same time bound.

Recently, Buss [bu87] devised an alternating log-time algorithm for both BSVP and the recognition problem for parenthesis context-free grammars. Since alternating log time is equivalent to $\mathbf{NC}^1$ [ru81] under a strong notion of uniformity, this finally settled the question of whether BSVP is in $\mathbf{NC}^1$. Buss's algorithm was based on converting the sentence into PLOF form (post-fix longer operand first) and then playing a two-person game on it. The game simulated the evaluation of the sentence and could be played in log-time on an alternating Turing machine. Buss also showed that his result is optimal in a very strong sense—he showed that BSVP is complete for alternating log-time under reductions from any level of the log-time hierarchy.

Dymond [dy88] extended Buss's result for parenthesis grammars to show that all input-driven languages can be recognized in $\mathbf{NC}^1$. His technique generalizes the game described by Buss.

Very recently, Muller and Preparata [mp88] devised log-depth circuits to solve formula evaluation for semi-rings. Their approach is based on using a universal evaluator to evaluate an infix formula where, for each operator, the longer operand occurs before

the shorter.

There are a number of reasons why the formula-value problem is interesting. The BSVP is the analogue of the circuit-value problem for which each gate has fan-out 1. The circuit-value problem is log space complete for P and hence is not in **NC** unless **P** equals **NC**. The BSVP, on the other hand, is clearly in **NC** and is, therefore, a natural candidate for an $\mathbf{NC}^1$-complete problem. Also, propositional formulas are fundamental concepts in logic, and the complexity of evaluating them is of interest.

In this paper, we present a simple $\mathbf{NC}^1$ algorithm for BSVP that incorporates Buss's original ideas into a two-person pebbling game similar to that introduced by Dymond and Tompa [dt85]. This algorithm is designed to give insight into the mechanism of Buss's algorithm. We show that our result is optimal by proving that the problem is complete for $\mathbf{NC}^1$ under $\mathbf{AC}^0$ reductions. We then proceed to use our evaluation technique to place the general arithmetic-formula evaluation problem over rings, fields, and semi-rings in arithmetic $\mathbf{NC}^1$.

This paper is organized as follows. The relevant background is given in §2. In §3 we describe an $\mathbf{NC}^1$ algorithm that translates Boolean sentences into PLOF sentences. In §4 an $\mathbf{NC}^1$ algorithm for the PLOF sentence-value problem is given. This finishes the proof that BSVP is in $\mathbf{NC}^1$. Some completeness results for BSVP are given in §5. In §6 we generalize the technique of §4 to obtain an arithmetic $\mathbf{NC}^1$ algorithm for arithmetic-formula evaluation (over rings, fields, and semi-rings).

2. Background.

2.1. Boolean circuit complexity.

All unreferenced material in this section is from [co85], and we refer the reader to that paper for a more in-depth discussion of Boolean circuit complexity.

DEFINITION. A *Boolean circuit* α on n inputs and m outputs is a finite directed acyclic graph with each node labeled from $\{x_1, \cdots, x_n, 0, 1, \neg, \vee, \wedge\}$. Nodes labeled x_i are *input nodes* and have indegree 0. Nodes with indegree 1 are labeled $\neg$, and those with indegree 2 are labeled either $\vee$ or $\wedge$, where each edge into the node is associated with one argument of the function corresponding to the label. There is a sequence of $m \geq 1$ nodes in α designated as *output nodes*. In practice, the nodes of α are called *gates*.

DEFINITION. For a circuit α, the *complexity* of α, designated $c(\alpha)$, is the number of nodes in α. The *depth* of α, designated $d(\alpha)$, is the length of the longest path from some input node to some output node.

We also assign to each gate in our Boolean circuits a gate number. We assume that in a given circuit α, each gate has a unique gate number and all gate numbers are between 0 and $c(\alpha)^{O(1)}$ (i.e., their binary encoding is $O(\log c(\alpha))$). Furthermore, we assume that all gates in a Boolean circuit are on a path from some input to an output. When the inputs are assigned values from $\{0, 1\}$, each output takes on a unique value. A circuit α on n inputs and m outputs computes a function $f : \{0,1\}^n \to \{0,1\}^m$ in the obvious way. We are interested in computing more general functions, namely those of the form $f : \{0,1\}^* \to \{0,1\}^*$. We need circuit families for this.

DEFINITION. A *circuit family* $\langle \alpha_n \rangle$ is a sequence of Boolean circuits such that the nth circuit in the family has n inputs and $h(n)$ outputs, where $h(n) = n^{O(1)}$.

Notice that arbitrary circuit families are very powerful (they can even recognize non-recursive languages). Therefore, we restrict ourselves to *uniform circuit families*. The strength of the Turing machines used to generate the circuits determines the uniformity condition on the circuit family. We note that all of our Turing machines are assumed to be multi-tape.

DEFINITION. Let α be a Boolean circuit. Let g be a gate in α and $\rho \in \{l, r\}^*$. Then $g(\rho)$ is the gate reached when ρ is followed (as a path) toward the inputs of α by starting at g. For example, $g(l)$ is g's left input. We make the convention that if g is an input gate, then $g(l) = g(r) = g$.

DEFINITION [ru81]. For a circuit family $\langle \alpha_n \rangle$, the *extended-connection language* L_{EC} consists of 4-tuples $\langle \bar{n}, g, \rho, y \rangle$, where $\bar{n} \in \{0,1\}^*$ ($\bar{n}$ is n in binary), $g \in \{0,1\}^*$ (g is a gate number), $y \in \{x_1, \cdots, x_n, 0, 1, \neg, \vee, \wedge\} \cup \{0,1\}^*$, and $|\rho| \leq \log c(\alpha_n)$ such that if $\rho = \epsilon$ then y is the label of the gate numbered g; otherwise, y is the gate number of $g(\rho)$. $\langle \alpha_n \rangle$ is U_E *uniform* if there is a deterministic linear-time Turing machine recognizing L_{EC}.

Here L_{EC} encodes local connection information in α_n, that is, connections that are within distance $\log c(\alpha_n)$.

Note. The time bound in Ruzzo's original definition of U_E uniformity is given as $O(\log c(\alpha_n))$. However, since the length of the input to the Turing machine is also $O(\log c(\alpha_n))$, our definition is equivalent—we prefer to use the size of the input to the machine in the definition.

DEFINITION. For all $k > 0$, define $\mathbf{NC}^k$ as the class of functions computable by a U_E-uniform circuit family $\langle \alpha_n \rangle$ such that $c(\alpha_n) = n^{O(1)}$ and $d(\alpha_n) = O(\log^k n)$. $\mathbf{NC} = \bigcup_{k>0} \mathbf{NC}^k$.

We use U_E uniformity in our definition of $\mathbf{NC}$ instead of the more common U_{E^*} uniformity. Ruzzo shows that $\mathbf{NC}^k$ ($k \geq 1$) is the same under both definitions. The advantage of using U_E uniformity is that the uniformity condition can be checked with the generally more familiar deterministic Turing machine (DTM) instead of an alternating Turing machine (ATM). The disadvantage is that ATMs are more powerful than DTMs and it may be easier to check the uniformity with an ATM.

Ruzzo [ru81] developed a Turing machine characterization of uniform Boolean circuits by showing that ATMs are basically uniform circuits.

PROPOSITION 2.1 [ru81]. *A problem is in* $\mathbf{NC}^k$ *if and only if it is solvable by an* ATM *in time* $O(\log^k n)$ *and space* $O(\log n)$.

Notice that in Proposition 2.1 the standard textbook definition of a Turing machine does not make sense because the time bound is sublinear (and, thus, not all of the input can be accessed on a single path of the computation). Therefore, we adopt the random-access multi-tape model described by Chandra, Kozen, and Stockmeyer [cks81]. This machine has a special index tape onto which the address of the input tape cell that needs to be accessed is written (in binary). The input head can then read the value of the input specified by this address. A further complication arises because circuit families are defined as computing multiple-valued functions (that is, the corresponding circuits may have more than one output gate), whereas Turing machines recognize sets of predicates. We make the convention that a Turing machine M is said to compute a function f if the predicate

$$A_f(c, i, z) \stackrel{\text{def}}{=} \text{the } i\text{th symbol of } f(z) \text{ is } c$$

is recognized by M.

Following Ruzzo [ru81], we also make a number of assumptions (without loss of generality) about ATMs. First, every configuration of an ATM has at most 2 successors. Second, all accesses to the ATM's input tape are performed at the end of the computation. This is easily accomplished by having the ATM guess the input and in parallel verify it (by looking at the input tape) and continue with the computation. Finally, we

make the convention that deterministic configurations are considered to be existential with one successor.

$\mathbf{NC}^1$ is considered to be a very fast complexity class, and many problems have been shown to be in $\mathbf{NC}^1$. Sum and product of 2 n-bit integers, sum of n n-bit integers, and sorting of n n-bit integers are all in $\mathbf{NC}^1$ [sa76]. Because of their shallow depth, $\mathbf{NC}^1$ circuits can always be converted into equivalent circuits with fan-out 1, polynomial size, and $O(\log n)$ depth. In this form, they can be expressed as formulas.

COROLLARY 2.2. $\mathbf{NC}^1$ *is the class of languages recognized by uniform log-depth formula families. The nth member of the family recognizes all strings in the language of length n.*

A generalization of the uniform Boolean circuit families are the unbounded fan-in uniform circuit families [csv82]. These circuit families are allowed arbitrary fan-in at the $\wedge$ and $\vee$ gates. We need a new uniformity condition.

DEFINITION. The *direct connection language* for an unbounded fan-in family of circuits $\langle \alpha_n \rangle$ (denoted L_{DC}) is given by the set of 3-tuples $\langle \bar{n}, g, y \rangle$, where $\bar{n}, g \in \{0,1\}^*$, $y \in \{x_1, \cdots, x_n, 0, 1, \neg, \vee, \wedge\} \cup \{0,1\}^*$ such that if $y \in \{0,1\}^*$ then y is an input to g; otherwise, y is g's label. $\langle \alpha_n \rangle$ is U_{DL} *uniform* if L_{DC} can be recognized by a DTM in linear space (i.e., $O(\log c(\alpha_n))$ space).

We define a hierarchy of unbounded fan-in circuits by the following.

DEFINITION. For all $k > 0$, $\mathbf{AC}^k$ is the class of problems solvable by an unbounded fan-in U_{DL}-uniform circuit family $\langle \alpha_n \rangle$, where $c(\alpha_n) = n^{O(1)}$ and $d(\alpha_n) = O(\log^k n)$. $\mathbf{AC} = \bigcup_{k>0} \mathbf{AC}^k$.

Once again we can characterize this hierarchy by using alternating Turing machines:

PROPOSITION 2.3 [co85]. *For all $k > 0$, $\mathbf{AC}^k$ is the class of problem solvable by an* ATM *in space $O(\log n)$ and alternation depth $O(\log^k n)$.*

The definitions above suffice to define $\mathbf{AC}^k$ when $k > 0$. However, we are interested in $\mathbf{AC}^0$ because we wish to show that BSVP is complete for $\mathbf{NC}^1$ under $\mathbf{AC}^0$ reductions. The uniformity condition is too strong in the circuit definition. The ATM definition (Proposition 2.3) would have to place further resource restrictions on the machine since a straightforward extension of the proposition would imply $\mathbf{AC}^0 = \mathbf{NL}$ (nondeterministic log space).

Immerman [im89] proposed defining $\mathbf{AC}^k$ ($k \geq 0$) in terms of a CRAM (a CRCW PRAM that is strengthened slightly to allow a processor to shift a word left or right by $\log n$ bits in unit time). This modification does not affect $\mathbf{AC}^k$ when $k > 0$. Immerman also gave a number of other characterizations of $\mathbf{AC}^0$, including first-order expressible properties and inductive definitions for which the depth of the induction is constant, and showed that all these characterizations are equivalent.

An alternate definition was proposed by Buss [bu87]. The log time hierarchy (denoted $\mathbf{LH}$) introduced by Sipser [si83] is the class of problems solvable by an ATM in log time and $O(1)$ alternations. Buss proposed $\mathbf{LH}$ as the definition of uniform $\mathbf{AC}^0$.

Recently Barrington, Immerman, and Straubing [bis88] showed that all 4 of the above characterizations give the same class, thus suggesting that these may be the appropriate definition. We would like a circuit definition of $\mathbf{AC}^0$. We begin by defining an appropriate uniformity condition.

DEFINITION [ru81]. $\langle \alpha_n \rangle$ is U_D *uniform* if its direct connection language, L_{DC}, can be recognized by a DTM in linear time (i.e., $O(\log c(\alpha_n))$ time).

Finally, $\mathbf{AC}^0$ is the class of problems solvable by an unbounded fan-in U_D-uniform circuit family $\langle \alpha_n \rangle$ that has constant depth and $n^{O(1)}$ size. This definition is consistent with the others due to the following.

THEOREM 2.4. *U_D-uniform* $\mathbf{AC}^0 = \mathbf{LH}$.

The proof of Theorem 2.4 requires the following lemma.

LEMMA 2.5. *If $L \in \mathbf{LH}$, then there is an* ATM *M that accepts L and there are $c, k \in \mathbb{N}$ so that for all n and all x in $\{0,1\}^*$ with $|x| = n$, and all configurations γ with $|\gamma| \leq c \log n$, every computation of M with input x starting in γ terminates within $c \log n$ steps and at most k alternations.*

To handle the time constraint, we incorporate a clock into the ATM that times the computation. When the clock runs out, the ATM automatically rejects. Because the clock can count in unary, this at most doubles the running time. For the bound on the alternations, we can use the finite state of the machine to count the number of alternations that have occurred.

Proof (*Theorem* 2.4). ($\subseteq$) An ATM M on input x guesses the output gate (say g) of $\alpha_{|x|}$. If g is "$\wedge$" ("$\vee$"), then M enters a universal (existential) state. M now guesses an input gate to g. If this gate is an input to the circuit, then M directly checks its corresponding input and accepts or rejects appropriately. Otherwise, it recursively applies this procedure to this new gate. All guesses about the circuit are verified by checking for the appropriate membership in L_{DC}. Since $\langle \alpha_n \rangle$ is constant depth and M uses at most 2 alternations to simulate each step of the circuit, M is in $\mathbf{LH}$.

($\supseteq$) Let $L \in \mathbf{LH}$. Let M be a log time ATM recognizing L, and let $c, k \in \mathbb{N}$, as in Lemma 2.5. We define a U_D-uniform family of circuits $\langle \alpha_i \rangle$ that simulate M and have depth k.

The gates of α_n are labeled by a 3-tuple $\langle \gamma, t, \rho \rangle$, where

1. γ is a configuration of M (on inputs of length n).
2. $t \in \{\wedge, \vee, I, \overline{I}, 0, 1\}$, where
   (a) $t = \wedge$ if γ is an universal configuration.
   (b) $t = \vee$ if γ is an existential configuration.
   (c) $t = I$ if γ is an input configuration and M in configuration γ accepts if $x_i = 1$, where i is on the index tape.
   (d) $t = \overline{I}$ if γ is an input configuration and M in configuration γ accepts if $x_i = 0$, where i is on the index tape.
   (e) $t = 0$ if γ is a rejecting configuration.
   (f) $t = 1$ if γ is an accepting configuration.
3. $\rho \in \{l, r\}^*$, where $|\rho| \leq c \log n$.

There is exactly one gate for every possible triple $\langle \gamma, t, \rho \rangle$. The output gate is the gate labeled $\langle \gamma_0, t_0, \Lambda \rangle$, where γ_0 is the initial configuration, t_0 is the type of γ_0, and Λ is the empty string. The input gates are the triples $\langle \gamma, t, \rho \rangle$ of type t equal to I, $\overline{I}$, 0 or 1; these are identified with the inputs x_i, $\overline{x}_i$, 0 and 1 (respectively), where i is the value on the index tape of the configuration γ. If $g_1 = \langle \gamma_1, t_1, \rho_1 \rangle$ and $g_2 = \langle \gamma_2, t_2, \rho_2 \rangle$ are gates of α_n, then g_1 is an input to g_2 if $t_1 \neq t_2$ and the computation described by ρ_1 starting in configuration γ_2 ends at γ_1 such that all configurations in this computation except the last are of type t_2.

It is straightforward to show by induction that a gate $g = \langle \gamma, t, \rho \rangle$ is 1 if and only if γ is accepting with respect to the input x. Also, the depth of the circuit is clearly the number of alternations of M.

It remains to be proved that the circuit family $\langle \alpha_i \rangle$ is U_D uniform. However, this now follows directly, since for the given gates $g_1 = \langle \gamma_1, t_1, \rho_1 \rangle$ and $g_2 = \langle \gamma_2, t_2, \rho_2 \rangle$ we can simulate the computation specified by ρ_1 to determine if g_1 is an input to g_2. □

Note. If the U_D-uniformity condition is used in the definition of $\mathbf{AC}^k$ ($k > 0$), the class does not change. Also, the ATM definition of $\mathbf{AC}^k$ (Proposition 2.3) can be

augmented to include $k = 0$ by adding the further restriction that the machine operate in time $O(\log^{k+1} n)$.

We are now ready to define $\mathbf{AC}^0$ reductions:

DEFINITION. Let A and B be sets. Then $A \leq_{\mathbf{AC}^0} B$ if there is a function f in $\mathbf{AC}^0$ such that for every x, $x \in A$ if and only if $f(x) \in B$.

THEOREM 2.6. *If* $A \leq_{\mathbf{AC}^0} B$ *and* $B \leq_{\mathbf{AC}^0} C$, *then* $A \leq_{\mathbf{AC}^0} C$.

Proof. Let f be an $\mathbf{AC}^0$ function such that $x \in A$ if and only if $f(x) \in B$. Let M be an ATM computing f. Let g be an $\mathbf{AC}^0$ function such that $x \in B$ if and only if $g(x) \in C$. Let N be an ATM computing g. We show that $g \circ f$ is computable in $\mathbf{AC}^0$ by an ATM T. Suppose the input to T is $\langle c, i, x \rangle$. T must accept if and only if the ith bit of $(g \circ f)(x)$ is c. T begins by simulating N on input $\langle c, i, x \rangle$ until N enters an input state. Suppose that when N does so, it has j written on its index tape and N would accept if x_j was $b \in \{0, 1\}$. T now finishes by simulating M on input $\langle b, j, x \rangle$. Clearly, T runs in log time, uses a constant number of alternations, and computes $g \circ f$. □

THEOREM 2.7. *If* $B \in \mathbf{NC}^1$ *and* $A \leq_{\mathbf{NC}^1} B$, *then* $A \in \mathbf{NC}^1$.

Proof. Let f be a function such that for every x, $x \in A$ if and only if $f(x) \in B$ and f is in $\mathbf{NC}^1$. Let M be an ATM recognizing B in log time. Let N be an ATM computing f (that is, recognizing A_f) in log time. We describe an ATM T that recognizes A. T simulates M except where M enters an input configuration. Suppose T has simulated M up to an input configuration and has i written onto its index tape and c as the guess for the ith input bit. At this point T begins to simulate N with the input $\langle c, i, x \rangle$. It is easy to see that T accepts input x if and only if M accepts $f(x)$. Also, since M and N run in log time, so does T. Therefore, $A \in \mathbf{NC}^1$. □

COROLLARY 2.8. *If* $B \in \mathbf{NC}^1$ *and* $A \leq_{\mathbf{AC}^0} B$, *then* $A \in \mathbf{NC}^1$.

2.2. Arithmetic circuit complexity. Most of the material in this section can be found in [jg86].

DEFINITION. An *arithmetic circuit (straight–line program)* over an algebraic structure F is a directed acyclic graph for which each node has indegree 0, 1, or 2. Nodes with indegree 0 are labeled as either input nodes or elements of F. Nodes with indegree 1 and 2 are labeled with the unary and binary operators of F, respectively. For example if F is a field, then the unary operators are "$-$" (additive inverse) and "${}^{-1}$" (multiplicative inverse) and the binary operators are "$+$" and "$\times$". There is a sequence of $m \geq 1$ gates with outdegree 0 designated as output nodes.

As with Boolean circuits, we assume there are no superfluous nodes. For an arithmetic circuit α the *complexity* and *depth* of α are defined the same as for Boolean circuits.

Arithmetic circuits are not sufficiently powerful for our purpose. For example, there may be no way to describe and manipulate the formula within the particular algebraic structure.

DEFINITION. An *arithmetic-Boolean circuit* over an algebraic structure F is an arithmetic circuit (over F) augmented with a Boolean component and an interface between the two. The Boolean component is a Boolean circuit. The interface consists of two special gates—sign : $F \to \{0, 1\}$, defined by $\text{sign}(a) = 0$ if and only if $a = 0$, and sel : $F \times F \times \{0, 1\} \to F$, defined by

$$\text{sel}(a, b, c) = \begin{cases} a & \text{if } c = 0, \\ b & \text{if } c = 1. \end{cases}$$

The definitions of complexity and depth for arithmetic-Boolean circuits are extended from arithmetic circuits. Also, the definitions of arithmetic-Boolean circuit families, uni-

formity, and parallel complexity classes (i.e., the **NC** hierarchy) are analogous to those for Boolean circuits.

Inputs to an arithmetic-Boolean circuit consist of algebraic values to the arithmetic circuit and Boolean values to the Boolean circuit. In the case of arithmetic-formula evaluation, the Boolean inputs will describe the structure of the formula and the arithmetic inputs will specify the value of the variables in the formula.

2.3. Problem definitions.

DEFINITION. A *Boolean sentence* is defined inductively by

1. 0 and 1 are Boolean sentences.
2. If α and β are Boolean sentences, then so are $(\neg\alpha)$, $(\alpha \wedge \beta)$, and $(\alpha \vee \beta)$.

The definition of Boolean sentences above describes sentences in infix notation. However, our algorithm will work with sentences in postfix (reverse Polish) notation with the further provision that for any binary operator, the longer operand occurs first.

DEFINITION. A *postfix-longer-operand-first* (PLOF) sentence is defined by

1. 0 and 1 are PLOF sentences.
2. If α and β are PLOF sentences where $|\alpha| \geq |\beta|$, then $\alpha\neg$, $\alpha\beta\wedge$, and $\alpha\beta\vee$ are PLOF sentences.

We define the value of a Boolean or PLOF sentence in the usual way, where 0 and 1 represent False and True, respectively.

DEFINITION. The Boolean sentence-value problem (BSVP) is as follows: Given a Boolean sentence A, what is the value of A?

DEFINITION [js82]. A *semi-ring* $\mathbb{S}$ is a 5-tuple $(S, \oplus, \otimes, 0, 1)$, where $0, 1 \in S$ such that

1. $(S, \oplus, 0)$ is a commutative monoid.
2. $(S, \otimes, 1)$ is a monoid.
3. $\otimes$ distributes over $\oplus$.
4. For every $a \in S, a \otimes 0 = 0 = 0 \otimes a$.

For convenience, we will also assume a unary operator "$\odot$," where $\odot a = a$ for every $a \in S$. This will give us flexibility to increase the size of a formula over a semi-ring.

Some examples of semi-rings are $\mathbb{S} = (\{0,1\}, \vee, \wedge, 0, 1)$, $\mathbb{S} = (\mathbb{Z}, \min, \times, +\infty, 1)$, and any ring $\mathbb{S}$.

DEFINITION. Let $\mathbb{S}$ be a semi-ring (which may also be a ring or field). An *arithmetic formula* over $\mathbb{S}$ with indeterminates $X_1, X_2, \cdots, X_n$, is defined by

1. For $1 \leq i \leq n$, X_i is an arithmetic formula.
2. For every $c \in \mathbb{S}$, c is an arithmetic formula.
3. If α is an arithmetic formula and θ is a unary operator of $\mathbb{S}$, then $(\theta\ \alpha)$ is an arithmetic formula.
4. If α and β are arithmetic formulas and θ is a binary operator of $\mathbb{S}$, then $(\alpha\ \theta\ \beta)$ is an arithmetic formula.

An arithmetic formula A with indeterminates $X_1, X_2, \cdots, X_n$ is denoted by $A(X_1, \cdots, X_n)$.

The Boolean formula discussed earlier is clearly a special case of these new formulas. We define postfix arithmetic formula and PLOF (postfix-longer-operand-first) arithmetic formulas to be exact analogs of their Boolean counterparts. The length of an arithmetic formula A (denoted $|A|$) is the number of nonparenthesis symbols in A (where an indeterminate is one symbol).

DEFINITION. Let $\mathbb{S}$ be a ring, field, or semi-ring. The *arithmetic-formula evaluation problem* is as follows: Given an arithmetic formula $A(X_1, X_2, \cdots, X_n)$ over $\mathbb{S}$ and constants $c_1, c_2, \cdots, c_n \in \mathbb{S}$, what is $A(c_1, c_2, \cdots, c_n)$?

2.4. Other definitions. Consider a Boolean sentence A. Define the *depth* of atoms of A as the level of nesting of parentheses in the subsentence containing the atom. We can view A as a binary tree, namely its (unique) parse tree, defined inductively as follows: the root is the operator of A of minimum depth and its children are the roots of the trees of the operands of the operator. Notice that we do not need parentheses in the tree representation. In exposition we will use the tree representation interchangeably with the infix or PLOF representation. Therefore, we carry over tree notions such as *root, child, ancestor* and *descendent* to sentences.

DEFINITION. Let A be a Boolean sentence. The length of A, denoted $|A|$, is the number of nonparenthesis symbols in A.

This definition has the desirable property that sentences have the same length regardless of the representation used (either infix or PLOF).

DEFINITION. Let A be a postfix Boolean sentence, and suppose $1 \leq j \leq k \leq n$. Then $A[j,k]$ is the string $A[j]A[j+1]\cdots A[k]$. The *subsentences* of A are those strings of the form $A[j,k]$ that form sentences. For $1 \leq k \leq n$, A_k denotes the unique subsentence of A of the form $A[j,k]$ for some j. We call A_k the subsentence *rooted at position* k or, for short, *rooted at* $A[k]$. We use $j \trianglelefteq k$ to mean that $A[j]$ is in A_k and $j \vartriangleleft k$ to mean $j \trianglelefteq k$ and $j \neq k$.

Note that $j \trianglelefteq k$ if and only if A_j is a substring of A_k if and only if $A[k]$ is an ancestor of $A[j]$. Also, the relation $\trianglelefteq$ forms a partial order. The following fact is used often:

LEMMA 2.9. *Let A be a postfix Boolean sentence $|A| = n$. Let $a, b, c < n$ such that $c \trianglelefteq a$, $c \trianglelefteq b$, and $a \leq b$. Then $a \trianglelefteq b$.*

Proof. The subsentence A_b is of the form $A[j,b]$ $(j < b)$. $c \trianglelefteq b \Rightarrow j < c < b$ and $c \trianglelefteq a \Rightarrow c < a$. Therefore, $j < c < a < b \Rightarrow a \trianglelefteq b$. □

DEFINITION. Let A be a Boolean sentence. Consider the sentence obtained by removing a subsentence A_k and replacing it with some constant c (i.e., $c \in \{0,1\}$). The resulting sentence is denoted by $A(k,c)$. We say that $A(k,c)$ is A with a *scar* at k and that $A(k,c)$ is *scarred*.

All the definitions made here can be translated to arithmetic formulas in a natural way, and we will use these definitions when discussing arithmetic formulas.

3. Translation of Boolean sentences to PLOF sentences. As a first step toward finding an $\mathbf{NC}^1$ algorithm for BSVP, we give an $\mathbf{NC}^1$ algorithm that translates Boolean sentences into PLOF sentences. It is well known [co85] that there is a uniform family of $\mathbf{NC}^1$ circuits in which the nth circuit computes the function

$$Count_n : \{0,1\}^n \rightarrow \{0,1\}^{\log n}$$

(i.e., given n Boolean values, output a binary string denoting their sum). Likewise, there is a uniform family of $\mathbf{NC}^1$ circuits where the nth circuit computes the summation of n n-bit numbers.

If A is an infix Boolean formula, let $A[i] \in \{\wedge, \vee, \neg, 0, 1\}$ be the ith nonparenthesis symbol in A. We describe an algorithm that outputs, for each $A[i]$, its position in the PLOF sentence.

DEFINITION. For A, an infix Boolean sentence, and $A[1]A[2]\cdots A[n]$, the enumeration of the nonparenthesis symbols of A, the *subsentence rooted at* $A[i]$ is the smallest subsentence of A containing $A[i]$. $A[k]$ is an *ancestor* of $A[i]$ if the subsentence rooted at $A[k]$ contains the subsentence rooted at $A[i]$.

LEMMA 3.1. *The following are computable in* $\mathbf{NC}^1$.

a. $Scope(A,i,j) \stackrel{\text{def}}{=} A[i]$ *in the subsentence rooted at* $A[j]$.

b. *For each j, the size of the subsentence rooted at $A[j]$.*

Proof. Notice that by counting the nonparenthesis symbols in A, it is easy to locate $A[j]$ in A.

a. Define the depth of nonparenthesis symbols in A in the normal way (i.e., with respect to the parse tree for A). Using $Count$, it is easy to determine the depth of each nonparenthesis symbol of A. This is used to find the parentheses that delimit the subsentence rooted at $A[j]$. $A[i]$ is in this subsentence if it sits between these parentheses.

b. Count the number of i for which $Scope(A, i, j)$ holds. □

To determine the position of nonparenthesis symbol $A[j]$ in the PLOF translation of A, do the following:

1. Calculate the size of the subsentence rooted at $A[j]$.
2. For each ancestor $A[k]$ of $A[j]$, let L_k (R_k) be the size of the subsentence rooted at the left (right) child of $A[k]$. Define

$$S_k = \begin{cases} L_k & \text{if } L_k \geq R_k \text{ and } Scope(A, j, \text{right child of } A[k]), \\ R_k & \text{if } R_k > L_k \text{ and } Scope(A, j, \text{left child of } A[k]), \\ 0 & \text{otherwise.} \end{cases}$$

3. The position of $A[i]$ is $\sum S_k$.

Because summation is $\mathbf{NC}^1$ computable, it is now easy to see that the function mapping an infix formula A to its PLOF translation is $\mathbf{NC}^1$ computable.

4. The algorithm for the PLOF sentence-value problem. Because there is an $\mathbf{NC}^1$ algorithm that translates a Boolean sentence into an equivalent PLOF sentence, it suffices by Theorem 2.7 to prove the next theorem in order to prove that the Boolean sentence-value problem is in $\mathbf{NC}^1$.

THEOREM 4.1. *There is an $\mathbf{NC}^1$ algorithm for determining the truth value of a* PLOF *sentence.*

For the remainder of this section, we present a proof of Theorem 4.1. A good way to explain the algorithm is to use an interpreted version of the Dymond–Tompa 2-person pebbling game [dt85] (this is the standard simulation of a Boolean circuit by an ATM). This version can be used to determine the output of a Boolean circuit C with specified inputs, as follows. The game has two players, called the `Pebbler` and the `Challenger`. The `Pebbler` has a supply of pebbles, each labeled either 0 or 1. The `Pebbler` moves by placing a pebble on a node of C. (The node is either a gate or an input to C.) The label on the pebble represents the `Pebbler`'s guess as to the value of the node pebbled. The `Pebbler` moves first by placing a pebble labeled 1 on the output node, representing a guess that the output value of the circuit is 1. After each `Pebbler` move, the `Challenger` moves by challenging some pebbled node. The challenged node must either be the one just pebbled or the node last challenged by the `Challenger`. The game ends when all inputs to the challenged node have been pebbled (pebbles are never removed once placed by the `Pebbler`). The `Pebbler` wins if and only if the label on the pebble of the challenged node is consistent with the node type and the labels on its inputs.

For example, if the challenged node is an input node with value 1, then the `Pebbler` wins if and only if the pebble on that node has label 1. If, on the other hand, the challenged node is an OR gate with pebble label c and its inputs have pebble labels x and y, then the `Pebbler` wins if and only if c is the logical OR of x and y.

LEMMA 4.2. *In the above game, the* `Pebbler` *has a winning strategy if and only if the circuit has output* 1.

Proof. If the circuit has output 1, then the `Pebbler`'s strategy is, for each move, to pebble with the correct label all unpebbled inputs to the challenged node. If the circuit has output 0, then the `Challenger`'s strategy is to always challenge the node of minimum depth whose pebbled value is incorrect. □

The above game forms the basis for our $\mathbf{NC}^1$ algorithm for determining the value of a PLOF sentence. The input to the algorithm is a PLOF sentence A, which is a string of symbols $A = A[1] \cdots A[n]$ from the alphabet $\{\vee, \wedge, \equiv, \neg, 0, 1\}$. Hence, the length of A, $|A|$, is n.

Without loss of generality, we may assume that n is a power of 2. If n is not in fact a power of 2, the algorithm proceeds as if a string of $\neg$'s is tacked on to the right end, bringing the length of the input to the nearest larger power of 2. If the number of such $\neg$'s is odd, the normal output of the algorithm is negated.

To adapt the pebbling game from the circuit C to the PLOF sentence A, the `Pebbler` places a pebble on a position k of A instead of a node of C, and the label on the pebble is a guess as to the value of subsentence A_k. The maximal subsentences of A_k are the inputs to the node.

If the sentence has value 1, then the `Pebbler` can force a win in $O(\log n)$ moves by the strategy used by Tompa [to85] to efficiently pebble a tree. (The basic idea goes back to Lewis, Stearns, and Hartmanis [lsh65] in their proof that context-free languages can be recognized in space $O(\log^2 n)$.)

This strategy can be described as follows: Consider the challenged subsentence A_k to be scarred by replacing each of its maximal pebbled subsentences by 0 or 1 (the label on the pebble). Place the next pebble on the subsentence A_j of A_k that comes as close as possible to cutting the scarred A_k in half. That is, the scarred size of A_j should be as close as possible to the new scarred size of A_k (in fact, the size of A_j will be between $\frac{1}{3}$ and $\frac{2}{3}$ the size of A_k). In this way, whether the `Challenger` next challenges the same position or the new position, the scarred size of the challenged subsentence is at least $\frac{1}{3}$ less after each pair of moves.

A straightforward implementation of this strategy on an ATM requires time $O(\log^2 n)$, since each of the $O(\log n)$ steps requires time $O(\log n)$ to describe a pebble position. To reduce the ATM time, we present a variation of the strategy that includes a uniform method for choosing subsentences, so that each pebble move can be described in a constant number of bits.

Associated with each of the `Pebbler`'s moves is a substring $g = A[i, j]$ of the input sentence A whose length is a power of 2. This substring includes the currently challenged position k and all unpebbled positions in the scarred subsentence A_k. All future moves are made within g. To help determine these moves, we define distinguished positions $V(g)$, $V_1(g)$, and $V_2(g)$, in g which depend only on A and the end points (i and j) of g.

DEFINITION. Let $g = A[i, j]$. If g has even length, define

$$V(g) = \max\{k | k \leq j \text{ and } i \trianglelefteq k\}.$$

That is, $A_{V(g)}$ is the maximal subsentence of A containing $A[i]$ whose root is in g. Further, define

$$V_1(g) = V(A[i, (i + j - 1)/2]) \quad \text{and} \quad V_2(g) = V(A[(i + j + 1)/2, j]).$$

Here, $V_1(g)$ and $V_2(g)$ are just "V(first half of g)" and "V(second half of g)," respectively.

LEMMA 4.3. *Let $g = A[i, j]$ have even length. Then $V(g)$ is one of $V_1(g)$ and $V_2(g)$.*

Proof. If $V(g) \leq (i + j - 1)/2$, then $V(g) = V_1(g)$; otherwise, it is $V_2(g)$. □

LEMMA 4.4. *Let $g = A[i, j]$ have even length. Then $V_1(g) + 1 \trianglelefteq V_2(g)$.*

Proof. If $V_1(g) = (i+j-1)/2$ or $V_1(g) \trianglelefteq V_2(g)$, the lemma is obvious. Therefore, assume that $V_1(g) < (i+j-1)/2$ and $V_1(g) \not\trianglelefteq V_2(g)$. Let $W = V(A[V_1(g)+1, j])$. We show that $W = V_2(g)$. If $W \geq (i+j+1)/2$, then $W = V_2(g)$ since A_W and $A_{V_2(g)}$ are maximal and both contain $A[(i+j+1)/2]$. Now suppose that $W \leq (i+j-1)/2$ (see Fig. 4.1).

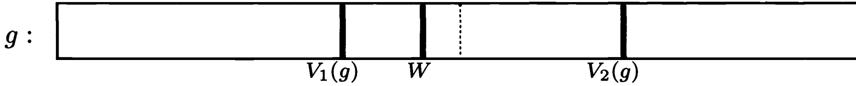


FIG. 4.1. $V_1(g) \not\trianglelefteq V_2(g)$ *and* $W \leq (i+j-1)/2$.

Clearly, A_W is the left operand of its parent operator, and its parent occurs to the right of g (since otherwise we could extend W to include its parent). But A_W is at least as long as its sibling to its immediate right because the input is a PLOF sentence. Since $|A_W| < (i+j-1)/2$, the entire sibling as well as the parent, must be in g, a contradiction to the definition of W. □

DEFINITION. Let $g = A[i, j]$ be a substring of A whose length $|g| = j-i+1$ is divisible by 4. Then g_1, g_2, and g_3 denote the left, middle, and right halves of g, respectively. That is, $g_1 = A[i, i+|g|/2-1]$, $g_2 = A[i+|g|/4, i+3|g|/4-1]$, and $g_3 = A[i+|g|/2, j]$.

Our pebbling game will allow the `Pebbler` to place pebbles only at the V position corresponding to each quarter of g (see Fig. 4.2).

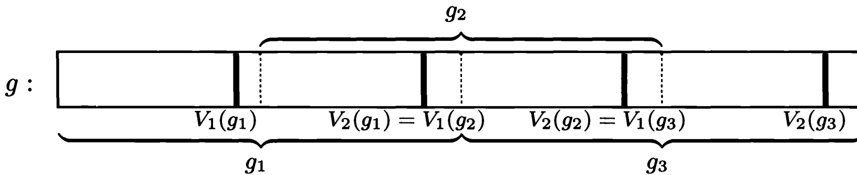


FIG. 4.2. *Pebble positions in g.*

It is easy to see

LEMMA 4.5. $V_1(g_1) < V_2(g_1) = V_1(g_2) < V_2(g_2) = V_1(g_3) < V_2(g_3)$.

Rules of the ATM game. The `Pebbler` may pebble up to 4 positions in one round of moves, as specified below. The `Challenger` challenges one of these 4. Associated with each round (except the first) is a substring g of A, whose length is a power of 2 and which contains the challenged position. We assume $|A| \geq 2$.

1. For the first round, the `Pebbler` places a pebble with label 1 on position n. The `Challenger` challenges n. The substring associated with the next round is $g = A[1, n]$. (Recall that n is a power of 2 by our earlier assumption.)
2. After the first round, the substring g contains the challenged position c. Assume $|g| \geq 4$. The pebbler considers $V_1(g_1), V_2(g_1), V_2(g_2), V_2(g_3)$, in that order, for pebbling. For each of these 4 candidates k, the `Pebbler` pebbles k if and only if $k \trianglelefteq c$ and no *pebbled* l satisfies $k \trianglelefteq l \triangleleft c$. The label on the pebble may be

either 0 or 1, except that if $k = c$ (the only case in which a position is allowed to be repebbled), the label must agree with the previous label. The `Challenger` challenges one such pebbled $V_i(g_j)$. Notice that by Lemma 4.3, this permits rechallenging c. The new substring is g_j.

3. Assume $|g| = 2$. Then g consists of the challenged position c and a second position k. The `Pebbler` pebbles k if and only if k is unpebbled and $k \lhd c$, and then repebbles c with the same label as before. The `Challenger` challenges one of the positions c' just pebbled. Lemma 4.6 below shows that all maximal subsentences of $A_{c'}$ have been pebbled. The `Pebbler` wins if and only if the pebble label on c' is consistent with the operator $A[c']$ and the pebble labels on the maximal subsentences of $A_{c'}$.

The following lemma justifies our assertion that if c is the challenged position, then the substring g contains all unpebbled positions in the scarred subsentence A_c. It also justifies the end condition in step 3 of the Rules.

LEMMA 4.6. *After each round in the* ATM *game, every position $k \lhd c$ with k to the left of g has some pebbled l such that $k \unlhd l \lhd c$.*

Proof. Induction on the round number.

Basis. Vacuous.

Induction. Suppose c' is the new challenged position and g_j is the new interval. By the Rules, $c' \unlhd c$. Let $k \lhd c'$, where k is to the left of g_j. Thus, $k \lhd c' \unlhd c$.

Case 1. k is to the left of g. By the induction hypothesis, there is a pebbled l with $k \unlhd l \lhd c$. By the *Rules*, $c' \ntrianglelefteq l$. But subsentences $A_{c'}$ and A_l both contain k, so by Lemma 2.9 one subsentence contains the other. Hence, $k \unlhd l \lhd c'$.

Case 2. k is in g but to the left of g_j. Then k is in g_1. If $j = 2$, then by Lemma 4.4 $k \unlhd V_1(g_1)$ or $k \unlhd V_2(g_1)$ and by the *Rules* $c' = V_2(g_2)$. Similarly, if $j = 3$, then $k \unlhd V_1(g_1)$, $k \unlhd V_2(g_1)$, or $k \unlhd V_2(g_2)$ and $c' = V_2(g_3)$. In general, say that $k \unlhd V_a(g_b)$. Then clearly $V_a(g_b) < c'$, and by Lemmas 2.9 and 4.5 $k \unlhd V_a(g_b) \lhd c' \unlhd c$ (since the subsentences at both position $V_a(g_b)$ and c' contain k). If $V_a(g_b)$ is pebbled, then we are done, with $l = V_a(g_b)$. Otherwise, by the *Rules*, there is a pebbled l with $V_a(g_b) \unlhd l \lhd c$. Again by the *Rules*, $c' \ntrianglelefteq l$. But subsentences $A_{c'}$ and A_l both contain $V_a(g_b)$, so one subsentence contains the other. Hence, $k \unlhd V_a(g_b) \unlhd l \lhd c'$. $\square$

LEMMA 4.7. *In the* ATM *game, the* `Pebbler` *has a winning strategy if and only if the value of A is* 1.

Proof. The proof is similar to that of Lemma 4.2. The positions pebbled are completely determined by the *Rules* and the positions challenged. The only choice given to the `Pebbler` is the labels on the pebbles. If the value of A is 1, the `Pebbler`'s strategy is to choose each label equal to the value of the subsentence pebbled. If the value of A is 0, the `Challenger`'s strategy is to challenge the leftmost incorrectly labeled pebble in each round. $\square$

It remains for us to show that the game can be implemented on an ATM in time $O(\log n)$.

LEMMA 4.8. *The following predicates are in* $\mathbf{NC}^1$.

a. $Subsentence(A, i, j) \stackrel{\text{def}}{=} A[i, j]$ *is a well-formed* PLOF *sentence.*

b. $Descendent(A, i, j) \stackrel{\text{def}}{=} i \unlhd j$.

c. $Child(A, i, j) \stackrel{\text{def}}{=} A_i$ *is a maximal proper subsentence of* A_j.

d. $V(A, i, j, k) \stackrel{\text{def}}{=} (k = V(A[i, j]))$.

Proof. Recall that the function *Count* is in $\mathbf{NC}^1$.

a. Check that the number of binary operators in $A[i,j]$ is one less than the number of constants and that for each binary operator in $A[i,j]$ (say $A[k]$) there are more constants in $A[i,k]$ than binary operators.

b. Find the unique k for which $Subsentence(A,k,j)$ holds, and check that $k \leq i \leq j$.

c. Assume that $A[j]$ is a binary operator and that $i \neq j-1$ (otherwise, it is trivial). Check that $A[i+1,j+1]$ is a sentence.

d. Since $V(g)$ is defined in terms of subsentences, descendents, and children, this is immediate from a.–c. □

LEMMA 4.9. *Let A be a* PLOF *Boolean sentence and $\langle p_1,\cdots,p_k\rangle$ ($p_i \in \{1,\cdots,4\}$ for all p_i) be a sequence representing the challenged positions (from among $V_1(g_1)$, $V_2(g_1)$, $V_2(g_2)$, and $V_2(g_3)$) in the first k rounds of a 2-player game as described above. Then, the following are in* $\mathbf{NC}^1$:

a. *Determining if the sequence is valid.*

b. *Determining the position of p_k in A.*

c. *Determining the interval g after the kth round.*

Proof. Let $|A| = n = 2^r$. We prove the above in the reverse order.

c. For each i ($1 \leq i \leq k$) let

$$L_i = \begin{cases} 0 & \text{if } p_i = 1,2, \\ \frac{1}{2}\cdot\frac{2^r}{2^i} & \text{if } p_i = 3, \\ \frac{2^r}{2^i} & \text{if } p_i = 4, \end{cases}$$

$$R_i = \begin{cases} \frac{2^r}{2^i} & \text{if } p_i = 1,2, \\ \frac{1}{2}\cdot\frac{2^r}{2^i} & \text{if } p_i = 3, \\ 0 & \text{if } p_i = 4. \end{cases}$$

Here L_i and R_i represent the amount the left and right boundaries of g are moved at the ith round of the game. Then, the current g is given by the string $A[1+\sum L_i, n-\sum R_i]$.

b. Let g' be the interval after the first $k-1$ moves (i.e., corresponding to plays $\langle p_1,\cdots, p_{k-1}\rangle$). Then, we use the predicate V to determine the position of the p_kth pebble placement at move k in g'.

a. Denote the currently challenged position after the first i moves by $I(\langle p_1,\cdots,p_i\rangle)$. For every i ($1 \leq i < k$) check that $I(\langle p_1,\cdots,p_i,p_{i+1}\rangle) \trianglelefteq I(\langle p_1,\cdots,p_i\rangle)$, and for every j ($1 \leq j < p_i$) check that $I(\langle p_1,\cdots,p_{i+1}\rangle) \ntrianglelefteq I(\langle p_1,\cdots,p_{i-1},j\rangle)$. All these checks can be performed in parallel using part b. to compute I. □

DEFINITION. Let A be a PLOF Boolean sentence. Then, a *k-round history* of A is a sequence $\nu = \langle \nu_1,\cdots,\nu_k\rangle$, where $\nu_i = \langle p_i, \tau_{i,1},\cdots,\tau_{i,p_i-1}\rangle$, $p_i \in \{1,\cdots,4\}$, and $\tau_{i,j} \in \{0,1\}$. A k-round history $\nu = \langle \nu_1,\cdots,\nu_k\rangle$ of A is *valid* if there is a play of the 2-person game outlined above such that each ν_i represents the ith round of this game.

By this we mean that for each round i ($1 \leq i \leq k$) p_i is the position (in the sequence $V_1(g_1), V_2(g_1), V_2(g_2), V_2(g_3)$) of the challenge node and $\tau_{i,j}$ is the value of the pebble placed by the `Pebbler` at the jth pebble position if this position was pebbled, and otherwise $\tau_{i,j}$ is arbitrary.

Using Lemmas 4.8 and 4.9, it is easy to prove the following.

LEMMA 4.10. *Let A be a* PLOF *Boolean sentence. Let ν be a k-round history of a game on A. Then, there is an* $\mathbf{NC}^1$ *algorithm to determine if ν is valid.*

Since $|g|$ is cut in half each round of the game, the number of rounds is at most $\log n + 1$. The ATM simulates the `Pebbler`'s moves by using existential states and the `Challenger`'s moves by using universal states. It records the history of the play in a string of 10 bits for each round. The 4 possible `Pebbler` moves are recorded by using a pair of bits each, telling (1) whether the position was pebbled and, if the position was pebbled, (2) the label. The `Challenger`'s move is recorded with 2 bits telling which of the potentially 4 moves is challenged. The finite-state control can ensure that the challenged position is one that was actually pebbled.

After the history of the play is recorded, the ATM checks whether 1) the `Pebbler`'s moves are legal and 2) whether the `Pebbler` won. The ATM accepts if and only if both conditions are true. To do 1), Lemma 4.10 is used. Condition 2) is checked by using the information in the history of the game and the *Child* predicate from Lemma 4.8. It is now easy to complete the proof of Theorem 4.1.

5. $\mathbf{NC}^1$ completeness of BSVP. Theorem 4.1 showed that there is an $\mathbf{NC}^1$ algorithm for recognizing true PLOF sentences; hence, by the $\mathbf{NC}^1$ translation of Boolean sentences into PLOF sentences there is an $\mathbf{NC}^1$ algorithm for recognizing true Boolean sentences. In this section we prove that these results are optimal.

THEOREM 5.1. BSVP *is* $\mathbf{NC}^1$ *complete under many-one* $\mathbf{AC}^0$ *reductions.* BSVP *is also* $\mathbf{NC}^1$ *complete under many-one deterministic log-time reductions.*

Recall that BSVP is the set of true Boolean sentences; Theorem 5.1 also holds for the set of true PLOF sentences. By Theorem 2.4, a deterministic log-time reduction also is an $\mathbf{AC}^0$ reduction. So to prove Theorem 5.1 it suffices to prove completeness under deterministic log-time reductions.

Thus, it suffices to exhibit, for a log-time ATM M, a deterministic log-time function f such that, for any input x (with $|x| = n$), $f(x)$ is a Boolean sentence that has value *true* if and only if M accepts x. The sentence $f(x)$ will essentially be the execution tree of M on input x where the $\vee$'s, $\wedge$'s, 0's, and 1's in $f(x)$ correspond to the existential, universal, rejecting, and accepting configurations in the execution of M, respectively. We begin by building the framework for the proof of Theorem 5.1.

Recall the assumptions made in §2.1 about ATMs—every configuration has at most 2 successors, all accesses to the ATM's input tape are performed at the end of the computation, and deterministic configurations are considered to be existential.

Let M be a log-time ATM; without loss of generality, the input alphabet for M is $\{0, 1\}$ and the runtime of M is bounded by $t(n) = c \cdot \log n + c$ on inputs of length n for some constant c. Throughout the remainder of this proof we will be working with this fixed M. For each configuration s of M, we denote by $l(s)$ and $r(s)$ the successor configurations of s, where the degenerate cases are defined by $l(s) = r(s)$ when s has exactly 1 successor and $s = l(s) = r(s)$ when s has no successors (i.e., s is a halt state or a read state).

Suppose s is a configuration and $\rho \in \{l,r\}^*$. Define

$$\rho(s) = \begin{cases} s & \text{when } \rho = \epsilon \text{ (the empty string)}, \\ \gamma(l(s)) & \text{when } \rho = l\gamma, (\gamma \in \{l,r\}^*), \\ \gamma(r(s)) & \text{when } \rho = r\gamma, (\gamma \in \{l,r\}^*). \end{cases}$$

Intuitively, ρ is a string of choices made by M, and $\rho(s)$ is the configuration of M reached from s by these choices.

We want to define a family of Boolean formulas $\langle F_n \rangle$ such that M on input x accepts if and only if $F_{|x|}(x)$ is a true Boolean sentence (here Boolean formulas are similar to the arithmetic formulas defined in §2.3 except they have $\wedge$ and $\vee$ as operators). Let I^M be the initial configuration of M. We define the nth Boolean formula F_n as follows: F_n has indeterminates $X_1, \cdots, X_n$. Let $\rho \in \{l,r\}^*$. First define the Boolean formulas $\beta_n(\rho)$ ($|\rho| \leq t(n)$) by

1. If $|\rho| = t(n)$, then $\rho(I^M)$ is a halting configuration and we define the following: If $\rho(I^M)$ is accepting (respectively, rejecting), then $\beta_n(\rho) = 1$ (respectively, $\beta_n(\rho) = 0$).
   If $\rho(I^M)$ is a read configuration with i on its index tape and M would accept (respectively, reject) if the ith bit of the input is 1, then $\beta_n(\rho) = X_i$ (respectively, $\beta_n(\rho) = \overline{X_i}$).
2. If $|\rho| < t(n)$, then let $\phi_l = \beta_n(\rho l)$ and $\phi_r = \beta_n(\rho r)$. If $\rho(I^M)$ is a universal configuration, then $\beta_n(\rho) = (\phi_l \wedge \phi_r)$ and otherwise $\beta_n = (\phi_l \vee \phi_r)$.

Now, $F_n = \beta_n(\epsilon)$. Clearly, $F_{|x|}(x)$ is *true* exactly when M accepts x.

If x is an input to M, then $x = x_1 \cdots x_n$ is a vector of 0's and 1's. We let $f(x)$ be the Boolean sentence obtained from $F_{|x|}$ by replacing each literal X_i by the binary digit x_i and each literal $\overline{X_i}$ by the binary digit $1 - x_i$. To prove Theorem 5.1, it will suffice to show that the function $f(x)$ is deterministic log-time computable. Recall that this means that there is a deterministic log-time Turing machine N that, on input $\langle x, i \rangle$, outputs the ith symbol of $f(x)$ in $O(\log n)$ time.

DEFINITION. Let ϕ be a Boolean formula. Let $|\phi|$ denote the number of symbols in ϕ, including parentheses. For each nonparenthesis symbol s in ϕ, the *height* of s is defined inductively by

1. If s is a 0 or 1 or for some i, X_i or $\overline{X_i}$, then the height of s is 0.
2. If s is an operator, then its height is 1 plus the maximum of the heights of its operands.

The height of ϕ is the maximum height over all the symbols of ϕ.

We notice that the formulas $\langle F_n \rangle$ are completely balanced (i.e., for every operator, both its operands have the same height). Also, F_n has height $t(n)$. It is easy to prove the following lemma by induction.

LEMMA 5.2. *Let ϕ be a completely balanced Boolean formula of height s with only binary connectives. Then $|\phi| = 2^{s+2} - 3$.*

Thus, $|F_n| = 2^{t(n)+2} - 3$, and for $\rho \in \{l,r\}^*$ ($|\rho| \leq t(n)$), $|\beta_n(\rho)| = 2^{s+2} - 3$, where $s = t(n) - |\rho|$. Our construction of the deterministic log-time algorithm is based on the following observations about F_n:

1. For each $i < 2^{t(n)+2} - 3$, there is a unique $\rho \in \{l,r\}^*$ such that the ith symbol of F_n is in $\beta_n(\rho)$ but not in $\beta_n(\rho l)$ or $\beta_n(\rho r)$. (We make the convention that the symbols of F_n and of $f(x)$ are numbered starting with 0.)
2. Given $\rho \in \{l,r\}^*$, there is a unique number N_ρ such that $\beta_n(\rho)$ occurs at positions $N_\rho, \cdots, N_\rho + |\beta_n(\rho)| - 1$ in F_n. Specifically, if $|\rho| < t(n)$,

(a) The leftmost "(" of $\beta_n(\rho)$ occurs at position N_ρ of F_n.
(b) The rightmost ")" of $\beta_n(\rho)$ occurs at position $N_\rho + |\beta_n(\rho)| - 1$.
(c) The root of $\beta_n(\rho)$ occurs at position $N_\rho + \frac{1}{2}(|\beta_n(\rho)| - 1)$.
(d) $\beta_n(\rho l)$ occurs at positions $N_\rho + 1, \cdots, N_\rho + \frac{1}{2}(|\beta_n(\rho)| - 3)$.
(e) $\beta_n(\rho r)$ occurs at positions $N_\rho + \frac{1}{2}(|\beta_n(\rho)| + 1), \cdots, N_\rho + (|\beta_n(\rho)| - 2)$.

3. Since $\beta_n(\rho)$ is completely balanced and has height $t(n) - |\rho|$, $|\beta_n(\rho)| = 2^{t(n)-|\rho|+2} - 3$. Hence, $N_{\rho l} = N_\rho + 1$ and $N_{\rho r} = N_\rho + 2^{t(n)-|\rho|+1} - 1$.

To compute the ith symbol of the Boolean sentence $f(x)$, we need to find a $\rho \in \{l, r\}^*$ such that $i = N_\rho$ or $i = N_\rho + 2^{t(n)-|\rho|+1} - 2$ or $i = N_\rho + 2^{t(n)-|\rho|+2} - 4$, which indicate that the ith symbol of F_n is the "(", the root, or the ")" of $\beta_n(\rho)$. It is then quite easy to simulate $M(x)$ to determine what the ith symbol of $f(x)$ is. We first give a naive algorithm for computing the ith symbol of $f(x)$; unfortunately, this naive algorithm does not execute in $O(\log n)$ time, so we shall later indicate how to improve its execution time.

Input: x, i
Output: The ith symbol of $f(x)$.
Step (1): Compute $n = |x|$.
Step (2): Compute $d = c \cdot \log n + c$. (This is easy because our logarithms are base two.)
Step (3): Check that $i < 2^{d+2} - 3 = |f(x)|$; if not, abort.
Step (4): Set $\rho = \epsilon$ (the empty string).
Set $s = d$.
Set $j = i$.
Step (5): (Loop while $s \geq 0$)
Select one case (exactly one must hold):
Case (5a): If $j = 0$, output "(" and halt.
Case (5b): If $0 < j < 2^{s+1} - 2$, set $j = j - 1$ and set $\rho = \rho\ell$.
Case (5c): If $j = 2^{s+1} - 2$, exit to step (6).
Case (5d): If $2^{s+1} - 2 < j < 2^{s+2} - 4$, set $j = j - (2^{s+1} - 2)$ and set $\rho = \rho r$.
Case (5e): If $j = 2^{s+2} - 4$, output ")" and halt.
Set $s = s - 1$.
If $s \geq 0$, reiterate step (5); otherwise, exit to step (6).
Step (6): Simulate M for $|\rho|$ steps to determine the configuration $\rho(I^M)$.
If $|\rho| < d$ and $\rho(I^M)$ is a universal configuration, output "$\wedge$".
Otherwise, if $|\rho| < d$, output "$\vee$".
Otherwise, if $\rho(I^M)$ is an accepting configuration, output "1".
Otherwise, if $\rho(I^M)$ is an rejecting configuration, output "0".
Otherwise, $\rho(I^M)$ is an input configuration with some number k written on the index tape. If the value of the ith symbol of x would cause this configuration to accept, output "1". Otherwise, output "0".

It should be clear by inspection that this algorithm correctly computes the ith symbol of $f(x)$. In an iteration of the loop in step (5), s is equal to $t(n) - |\rho|$, and it has already been ascertained that the ith symbol of $F_{|x|}$ is the jth symbol of the subformula $\beta_{|x|}(\rho)$. The subformula $\beta_{|x|}(\rho)$ is of the form $\beta_{|x|}(\rho l) * \beta_{|x|}(\rho r)$, where $*$ is either $\vee$ or $\wedge$; the five cases correspond to the jth symbol being (a) the initial parenthesis, (b) in the subformula $\beta_{|x|}(\rho l)$, (c) the logical connective symbol, (d) in the subformula $\beta_{|x|}(\rho l)$, or (e) the final parenthesis.

To complete the proof of Theorem 5.1, we must prove that there is a log-time deter-

ministic Turing machine N for computing the ith symbol of $f(x)$. In the above algorithm, each step other than step (5) takes $O(\log n)$ time. In particular, for step (6) the simulation of M is hard wired and N simulates each operation of M with only one operation. Step (1) can be executed by finding the least k such that $n < 2^k$ and then using a binary search to calculate n. Step (5), however, is more difficult: There are $O(\log n)$ iterations of the loop, and each iteration takes $O(\log n)$ time in our naive implementation—we need each iteration to take constant time.

The reason that each iteration takes $O(\log n)$ time is that in case (5d), for example, to subtract $2^{s+1} - 2$ from j, both the high- and low-order bits of j must be modified; but j has $O(\log n)$ bits, so it takes too much time just to move the tape head from one end of j to the other. Similar problems arise in comparing j to $2^{s+1} - 2$ and $2^{s+2} - 4$. Also, even when just decrementing j by 1 in case (5b), it may take $O(\log n)$ time to propagate a borrow.

Fortunately, all these problems can be avoided by a simple trick. Before starting step (5), N breaks j into two parts: the low-order $2 + \log d$ bits of j are stored on a tape in *unary* notation; the remaining high-order bits of j are kept on a different tape in binary notation. Thus, to decrement j by 1, N merely changes one tape square on the unary tape and moves that tape head one square. To subtract $2^{s+1} - 2$ from j, N need only change two squares on the unary tape and modify one square of the binary tape (since $j \leq 2^{s+2} - 4$). A complication arises when there is a carry or borrow out of the $(2 + \log d)$-th bit position of j. N handles this by allowing the unary tape to overflow (and cause a carry) or underflow (and cause a borrow). To do this the unary tape is initialized with a marker indicating where the overflow or underflow occurs; since the unary part of j is changed by -1 or $+2$ at most $d = c \cdot \log n + c$ times, at most one marker is needed. During the iterations of the loop in step (5) N remembers whether or not an underflow/overflow has occurred. N also initializes the binary tape with a marker that indicates how far the borrow or carry will propagate.

We can now summarize how N executes step (5) in $O(\log n)$ time. First j is split into binary high-order and unary low-order parts—these are stored on separate tapes along with borrow/carry information. Then the loop is executed for $s = d$ to $s = 1 + \log d$, maintaining the value of j in the split binary/unary form. After these iterations, the higher-order, binary portion of j is equal to zero. The unary portion of j is now converted back to binary notation, and the remaining iterations of the loop with $s = \log d$ to $s = 0$ are executed in the normal naive fashion with j in binary notation. This completes the proof of Theorem 5.1. □

The set of true PLOF sentences is also complete for $\mathbf{NC}^1$ under deterministic log-time reductions. This is proved similarly to the proof of Theorem 5.1: It must be shown that the ith symbol of $f(x)$ in postfix notation can be obtained in deterministic log time.

6. Log depth circuits for arithmetic formula evaluation. We begin by describing a 2-player game (similar to that in §4) for evaluating arithmetic formulas over commutative semi-rings. We then transform this game into a log depth arithmetic-Boolean circuit over the commutative semi-ring. Finally we show how the game can be modified to solve the problem for noncommutative semi-rings, rings, and fields.

Throughout this section let $\mathbb{S}$ be some fixed commutative semi-ring and A be an arithmetic formula over $\mathbb{S}$ of length n. Without loss of generality we can assume that n is a power of 2. If n is not a power of 2, we assume that A has a string of $\odot$ attached to the left-hand side, bringing the total length of A to the next power of 2. (Recall that "$\odot$" is the unary identity operator.) Let $A(j, X)$ be A with A_j (the subformula rooted at position j) replaced by the indeterminate X. Recall that this is equivalent to saying that

A is scarred at j. Then we can write

$$A(j,X) = B \cdot X + C \qquad (B, C \in \mathbb{S}).$$

Therefore, determining the value of A can be broken into 3 subproblems: Evaluate A_j for some appropriately chosen j, determine B, and determine C. This procedure can recursively be applied to evaluate A_j. However, if we now apply this procedure to $A(j,X)$, we end up with the formula $[A(j,X)](j',X')$, where $A[j']$ is not necessarily an ancestor of $A[j]$. That is, the new formula may have 2 scars. After $O(\log n)$ steps, the formula can end up with $O(\log n)$ scars, making the procedure useless.

Brent [br74] solved this problem by allowing only one scar in any formula. In his algorithm, A is initially scarred by a subformula of A (say A_j) of size approximately $|A|/2$. A_j is handled recursively. However, the next scar of A is chosen so that its root is an ancestor of j. Therefore, at any step in the algorithm the subformula being evaluated has at most 1 maximal scar. A straightforward implementation of this technique would require $O(\log^2 n)$ time, since finding successive j's takes $O(\log n)$ time and the algorithm takes $O(\log n)$ rounds.

We modify the pebbling game of §4 to maintain the condition of having only 1 scar. In this new game pebbles have no labels and pebbles can be removed as well as added. The game ends with a win for the pebbler when all inputs of the challenged node are pebbled. Suppose that the pebbler has a strategy such that after every challenge (after some pebbles are possibly removed), each pebbled subformula has at most 1 maximal scar unless both children of the subformula are pebbled. As before, in the first round the pebbler pebbles the root of the input formula and the challenger challenges this node; in each subsequent round the challenger challenges a node pebbled in the current round or rechallenges the node challenged in the previous round. If the pebbler has an r round winning strategy on any play on a given input formula, then the following theorem shows that there is a circuit of depth $2r$ that computes the value of the formula. The resulting circuit family may not be uniform, and later we describe a strategy that can be implemented on a uniform circuit family.

THEOREM 6.1. *Let A be an arithmetic formula, and let the pebbler in the above 2-person game have an r round winning strategy on all plays on A. Then there is a circuit of depth $2r$ that computes the value of A.*

Proof. We prove the more general result that if the pebbler has an r round winning strategy on all plays on a formula A with a scar X at position i, then a circuit of depth $2r$ suffices to compute values B, C such that

$$A(i,X) = B \cdot X + C.$$

The result required in the theorem is simply the special case in which A has no scar; in this case $B = 0$ and C is the value of A.

The proof is by induction on the number of pebble moves. The base case $r = 0$ is straightforward and is omitted. Assume inductively that any formula with a single maximal scar for which the pebbler has an $r-1$ round winning strategy on all plays can be computed by a circuit of depth $2(r-1)$. Let A be a formula with a single maximal scar X at position i, and let the pebbler have a winning r round strategy on all plays on this formula. Let $|A| = n$. We now show that there is a circuit of depth $2r$ that computes the value of $A(i,X)$.

In the first round of the game the position n is pebbled and challenged as required. Consider the next move by the pebbler. If this move does not provide a new scar for A, then the pebbler has an $r-1$ round winning strategy on all plays of A. Hence, A can be

evaluated by a circuit of depth $2(r-1)$ by the induction hypothesis, and we are done. If the move does provide a new scar Y at position j for A we distinguish two cases: 1) A has two maximal scars X and Y, and 2) Y is an ancestor of X and hence A continues to have one maximal scar.

Case 1. If X and Y are two distinct maximal scars of A, then by assumption X and Y are the two children of A. Since it is possible for position j to be challenged in the current round, the pebbler has an $r-1$ round winning strategy for any play on the subformula rooted at position j. Hence, by the induction hypothesis, there is a circuit of depth $2(r-1)$ that computes the value of Y. But $A(i,X) = B\cdot X+C$ with $B = 1$ and $C = Y$ if the root of A is an addition node, and $B = Y$ and $C = 0$ if the root of A is a multiplication node. Hence, $A(i,X)$ can be computed by a circuit of depth $2(r-1)$ in this case.

Case 2. The new scar Y is an ancestor of the old scar X (see Fig. 6.1).

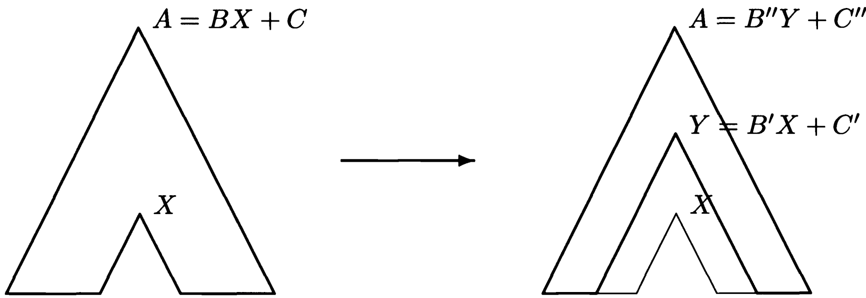


FIG. 6.1. *New scar Y is an ancestor of old scar X.*

As in Case 1, it is possible for position j to be challenged in the current round, hence the pebbler has an $r-1$ round winning strategy for the formula Y with a single maximal scar X. Hence, we have circuits to compute B' and C', each of depth $2(r-1)$, such that $Y(i',X) = B'\cdot X + C'$, where i' is the new position of i in the formula Y. Similarly, it is possible for position n to be rechallenged in the current round, so the pebbler has an $r-1$ round winning strategy for the formula A with a single maximal scar Y. Hence, we have circuits of depth at most $2(r-1)$ to compute B'' and C'' such that $A(j,Y) = B''\cdot Y + C''$. But $A(i,X) = B''(B'\cdot X + C') + C'' = B\cdot X + C$, giving

$$B = B''\cdot B' \quad \text{and} \quad C = B''\cdot C' + C''.$$

Since a circuit of depth 2 computes B and C in terms of B', C', B'', and C'', a depth $2r$ circuit suffices to compute $A(i,X)$.

This completes the induction step, and the theorem is proved. □

We now show how to make the game uniform.

DEFINITION. Let A be a PLOF formula, $|A| = n$. For $i,j < n$, the *least common ancestor* of i and j (denoted $lca(i,j)$) is the common ancestor of i and j with minimum depth. Furthermore, $right(i)$ denotes the right child of node i.

In our new game, we add 5 pebbling points to the 4 used in the Boolean game of §4. The object will be to ensure that the challenged formula either is contained in the new

interval or has a leftmost scar in the interval. In the original game, the pebbled positions in an interval g were $V_1(g_1)$, $V_2(g_1)$, $V_2(g_2)$, and $V_2(g_3)$. We augment these with

1. $lca(V_1(g_1), V_2(g_1))$ denoted by $Lca_1(g)$.
2. $right(Lca_1(g))$ denoted by $R_1(g)$. This is a pebble position only if $Lca_1(g) \neq V_2(g_1)$.
3. $lca(V_2(g_1), V_2(g_2))$ denoted by $Lca_2(g)$.
4. $right(Lca_2(g))$ denoted by $R_2(g)$. Again, this is a pebble position only if $Lca_2 \neq V_2(g_2)$.
5. The node challenged in the previous round denoted by $last(g)$.

In this game we explicitly include the challenged node from the previous round since it may not be one of the other 8. Figure 6.2 shows one possible placement of pebbles.

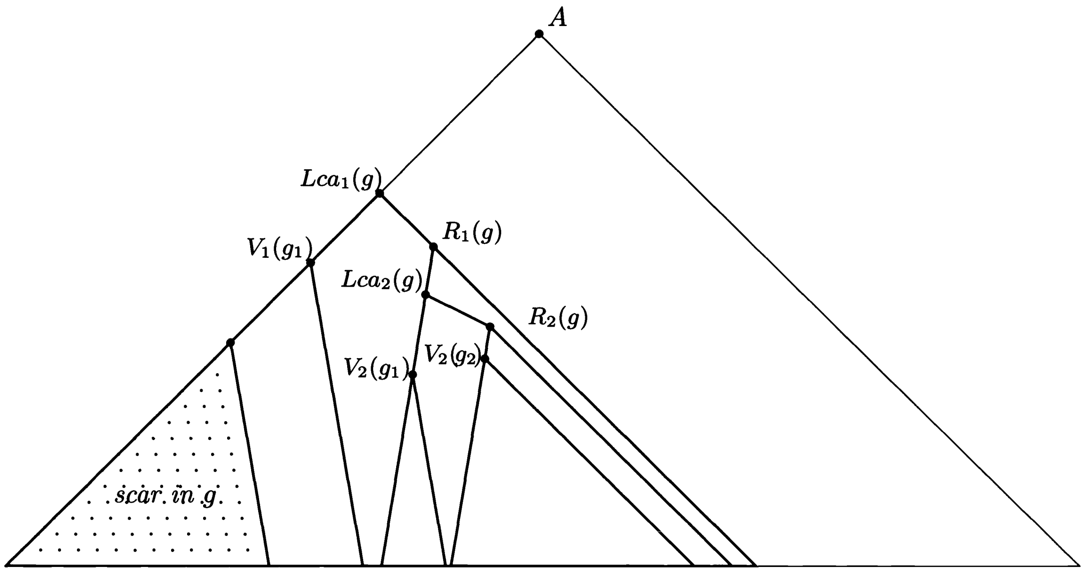


FIG. 6.2. *One possible pebble placement.*

DEFINITION. Let A be a PLOF formula. A_k is a *leftmost* subformula of A if it is a subformula of A such that when A is viewed as a tree, A_k occurs along the leftmost branch. A *leftmost scar* of A is a maximal pebbled leftmost subformula.

Notice that if F is a subformula of a PLOF A and F_k is a leftmost subformula of F, then F_k is an initial segment of F. The following rules ensure that every challenged formula has a single leftmost scar.

Rules of the algebraic game. Let A be an arithmetic formula, $|A| = n$ a power of 2 ($n \geq 2$).

1. In the first round, the `Pebbler` places a pebble on n, and the `Challenger` challenges it. In all subsequent rounds there will be an interval g whose length is a power of 2 and a challenged position c within g. For the next round, $g = A[1, n]$ and $c = n$.
2. For c, the challenged position in in g, if at least one child of c is not pebbled, then let $\nu_1, \cdots, \nu_9$ be the 9 pebble positions defined earlier such that $\nu_1 \leq \cdots \leq \nu_9$. We consider these for pebbling in this order. For each of these 9 candidates ν_i,

the `Pebbler` pebbles ν_i if and only if $\nu_i \trianglelefteq c$ and there is no pebbled l satisfying $\nu_i \trianglelefteq l \triangleleft c$. The `Challenger` challenges one of these new pebble positions. Notice that this allows rechallenging c. After the round ends all newly placed pebbles except the challenged node and any leftmost scar are removed, unless both children of the challenged node are pebbled. The new substring is the leftmost g_j containing the new challenged position.

3. If both children of c are pebbled, then the `Pebbler` wins.

LEMMA 6.2. *Let g be the current interval, and let c be the challenged node. Then either* 1) *A_c is contained in g or* 2) *A_c has a leftmost scar in g.*

Proof. We proceed by induction on the round number. Notice that to establish condition 2, it suffices to show that some leftmost subformula of A_c rooted in g is pebbled.

Basis. At round 1, $A = A_c$ satisfies condition 1.

Induction. In general, suppose the lemma holds for an interval g and challenged node c. If the new challenged node c' is in g_1, then g_1 is the new interval and either condition 1 holds for c' and g_1 or condition 2 holds with the same scar as for A_c.

Now suppose c' is in the second half of g_2, so that g_2 is the new interval, and suppose that $A_{c'}$ is not contained in g_2. Assume that the leftmost scar of A_c does not lie in g_2 (since otherwise we are done). Therefore, it lies in the first half of g_1. There are two subcases, depending on whether $A_{c'}$ includes an initial segment of g. If it does, then the leftmost scar of A_c is a leftmost subformula of $A_{c'}$, and $Lca_1(g)$ is either a leftmost scar of $A_{c'}$ in g_2 (so condition 2 holds for c' and g_2) or $Lca_1(g)$ is c', in which case the two children of c' are $V_1(g_1)$ and $R_1(g)$, so the game ends. The second subcase is that $A_{c'}$ is not an initial segment of g (see Fig. 6.3).

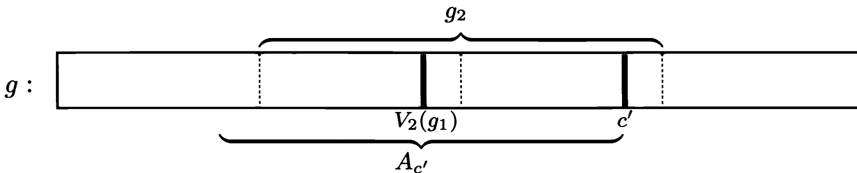


FIG. 6.3. *Subcase in which $A_{c'}$ is not an initial segment of g.*

In that case, $V_2(g_1)$ is a leftmost subformula of $A_{c'}$ because of the PLOF property.

Finally, suppose c' is in the second half of g_3, so that g_3 is the new interval. Furthermore, assume that $A_{c'}$ does not lie entirely in g_3. If the left child of c' is to the left of g_3, then it is in g_1 (since A_c is in g or has a leftmost scar in g), so either $c' = Lca_1(g)$ or $c' = Lca_2(g)$. In either case, both children of c' are pebbled, so the game ends. If the left child of c' is in the left half of g_3, it is $V_2(g_2)$ and condition 2 holds for c' and g_3. The final case occurs when the left child of c' is in the right half of g_3. Then one of $\{Lca_1(g), Lca_2(g), V_2(g_2)$, the leftmost scar of $c\}$ provides a leftmost scar of c' in g_3. □

Lemma 4.6 can easily be adapted to show that in any round of the game, every position $k \triangleleft c$ with k to the left of g has some pebbled l such that $k \trianglelefteq l \triangleleft c$.

LEMMA 6.3. *In the game, the indicated strategy for the* `Pebbler` *wins in $O(\log n)$ rounds.*

Proof. In every round of the game, the interval g is cut in half. □

We must now show that the game can be converted into a uniform log-depth arithmetic-Boolean circuit family.

In the game above, we did not know where any Lca_i, R_i, or $last$ was in the interval g. If one of these positions was challenged, a constant depth circuit could not determine the new interval. We modify the game slightly so that the `Pebbler` must specify an interval when a pebble is placed (corresponding to the interval the pebble is in). This interval can be placed as a label on the pebble. This gives a total of at most 14 different pebble points and labels (since many of the pebble points cannot be in every interval).

We must augment Lemma 4.8 to show that the new pebbling points can be determined in Boolean $\mathbf{NC}^1$. However, although the least common ancestor (and its right child) can be determined once the interval is known, the entire history of the game may be necessary to determine the last challenged position. Let $\nu = \langle p_1, \cdots, p_k \rangle$ ($p_i \in \{1, \cdots, 14\}$) be a sequence that describes the first k moves of the game. Here, p_i denotes the pebble challenged by the `Challenger` in the ith round.

LEMMA 6.4. *The following predicates are in Boolean* $\mathbf{NC}^1$:

a. $Lca_1(A, i, j, k) \stackrel{\text{def}}{=} (k = Lca_1(A[i, j]))$.

b. $Lca_2(A, i, j, k) \stackrel{\text{def}}{=} (k = Lca_2(A[i, j]))$.

c. $R_1(A, i, j, k) \stackrel{\text{def}}{=} (k = R_1(A[i, j]))$.

d. $R_2(A, i, j, k) \stackrel{\text{def}}{=} (k = R_2(A[i, j]))$.

e. $Last(A, i, j, k, \nu) \stackrel{\text{def}}{=}$ *k is the challenged node in the previous round of the game.*

Proof.

a., b. By Lemma 4.8, we can determine $V_1(A[i, j])$ and $V_2([A(i, j)])$ in $\mathbf{NC}^1$. The *Descendent* predicate in Lemma 4.8 can be used to check that one of the V_i occurs as a descendent of the left child of k and the other as a right child.

c., d. Determining the right child of a least common ancestor is easy once we can determine the least common ancestor.

e. Let $\nu = \langle p_1, \cdots, p_k \rangle$. In ν, find the largest i such that p_i is not $last$ (say p_l). Find the indicated pebble position in round l. This will be the challenged position in the current round. □

We can use a slightly modified Lemma 4.9 to determine if $\nu = \langle p_1, \cdots, p_k \rangle$ codes a valid sequence of challenges.

Let A^ν denote the scarred subformula challenged after the k indicated challenges, and let $I(\nu)$ be the position of the root of A^ν. Let Ω_ν be the circuit that computes the value of A^ν and α_ν be the value computed by the circuit Ω_ν. To obtain the desired log-depth arithmetic-Boolean circuit over the commutative semi-ring $\mathbb{S}$ we must compute α_ν with a constant-depth circuit using the values $\alpha_{\nu^\frown\langle q \rangle}$ ($q \leq 14$). We break Ω_ν into subcircuits $\Omega_{\nu^\frown\langle q \rangle}$. α_ν will be either a single value in $\mathbb{S}$ or a tuple, depending upon which of the cases in Lemma 6.2 holds. We will describe the circuit for the case where A^ν has a leftmost scar in g. The other case is an easy modification of this. Figure 6.4 is a simplified circuit to compute the value of A^ν for the pebble placements in Fig. 6.2.

Suppose that A^ν has $A^{\nu'}$ as a leftmost scar. Denote by σ_q the value $\alpha_{\nu^\frown\langle q \rangle}$, assuming the correct values (or tuples) at all pebble positions in this round of the game that came before the qth are computed. Now our algorithm is as follows.

Algorithm: *Compute α_ν where A^ν satisfies case* 2 *of Lemma* 6.2

$q_{\min}$:= `smallest` q `satisfying` $I(\nu') < I(\nu^\frown\langle q \rangle) < I(\nu)$

$q_{\max}$:= `largest` q `satisfying` $I(\nu') < I(\nu^\frown\langle q \rangle) \leq I(\nu)$

`In parallel, for each` $q \in q_{\min}, \cdots, q_{\max}$ `compute:` $\alpha_{\nu^\frown\langle q \rangle}$

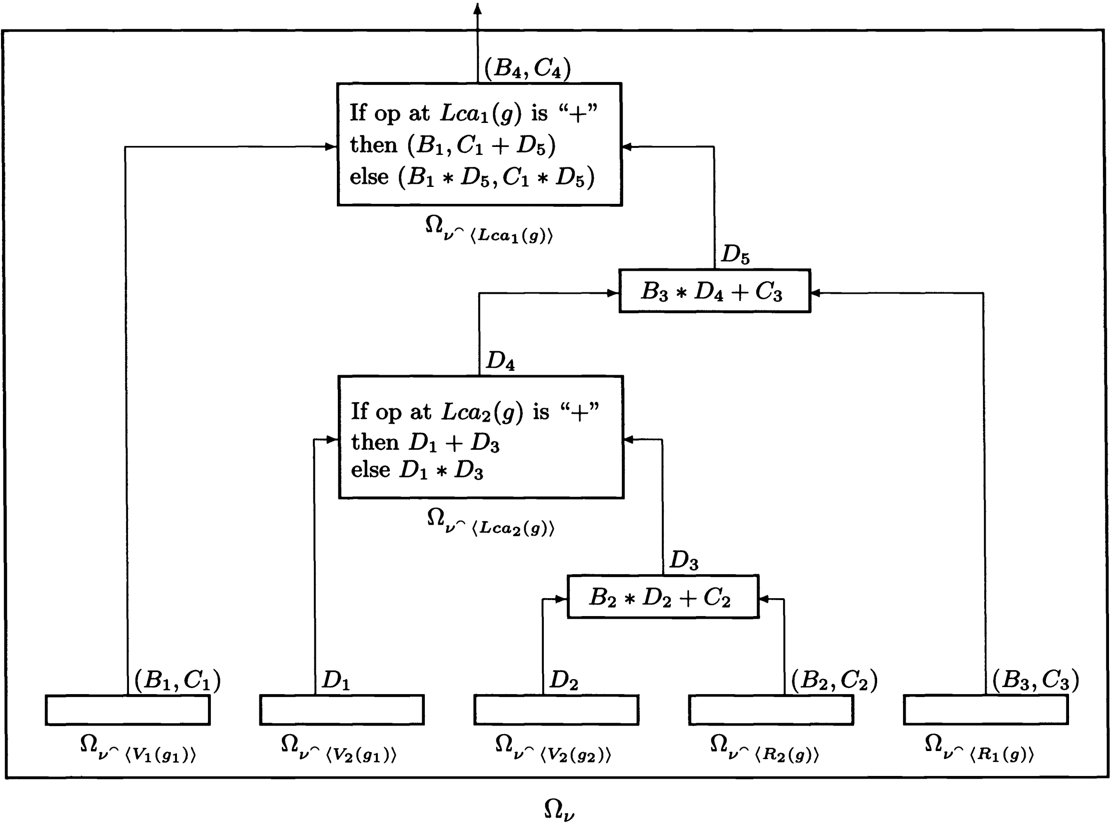


FIG. 6.4. *Circuit for computing A^v for pebble placements in Fig. 6.2.*

$\sigma_{\min} := \alpha_{\nu^\frown \langle q_{\min}\rangle}$
For $q = q_{\min} + 1$ to $q_{\max}$
    Let $\mu = \nu^\frown \langle q\rangle$
    If A^μ satisfies condition 1 of Lemma 6.2 then $\sigma_q := \alpha_\mu$
    else if A^μ satisfies condition 2 of Lemma 6.2 then
        Let $(B, C) = \alpha_\mu$
        Let q_1 be the position of the maximal leftmost pebbled
            subformula of A^μ
        If σ_{q_1} is an element of $\mathbb{S}$ then $\sigma_q := B * \sigma_{q_1} + C$
        else Let $(B', C') = \sigma_{q_1}$
            $\sigma_q := (B * B', B * C' + C)$
    else Let q_1 and q_2 be the pebble placements of the left
            and right operands of A^μ
        Let θ be the operator at A^μ
        If σ_{q_1} is an element of $\mathbb{S}$ then $\sigma_q := \sigma_{q_1} \theta \sigma_{g_2}$

```
        else Let (B, C) = σ_{q1}
                If θ = * then σ_q := (B * σ_{q2}, C * σ_{q2})
                else σ_q := (B, C + σ_{q2})
    Endfor
```

The only change that must be made to the algorithm for it to work for the other condition of Lemma 6.2 above is the value of $q_{\min}$.

The technique described above can easily be generalized to solve the problem for fields and (noncommutative) semi-rings. We show the field case first. Suppose $\mathbb{F}$ is a field and A is an arithmetic formula over $\mathbb{F}$. It is easy to show that a scarred formula $A(j, X)$ can be written as a rational affine function:

$$A(j, X) = \frac{B \cdot X + C}{D \cdot X + E} \qquad (B, C, D, E \in \mathbb{F}).$$

It is also easy to verify that these functions are closed under composition. Therefore, the same algorithm as above is used except the value α_ν of a subformula with a leftmost scar is represented by a 4-tuple (B, C, D, E).

For the (noncommutative) semi-ring case, we must first convert to PLOF form. However, we have problems if $*$ is not commutative. Therefore, we augment the language to include a "reverse multiplication," denoted $*'$, where $a * b = b *' a$. Any formula can be put in equivalent PLOF form in this augmented language. Now, a scarred formula $A(j, X)$ can be written as

$$A(j, X) = B \cdot X \cdot C + D \qquad (B, C, D \in \mathbb{S}).$$

A subformula $A(j, X)$ is represented by a 3-tuple (B, C, D). Again, composition is easy to do. Therefore, the semi-ring algorithm can be used except that a little care is necessary in keeping the left and right multipliers separate.

Finally, we present a simpler method for solving the evaluation problem when the algebra is a ring. Suppose we wish to evaluate the scarred formula $A(j, X) = B \cdot X + C$. Then, $A(j, 0) = C$ and $A(j, 1) = B + C$. From this system of equations we can easily determine both B and C. Therefore, the problem of determining A is broken into three subproblems: Evaluate the formula rooted at X, evaluate $A(j, 0)$, and evaluate $A(j, 1)$. These problems can be recursively solved.

Acknowledgments. We thank Martin Tompa for helping with the exposition in §4. We also thank the referees for their careful reading of the manuscript and many useful suggestions.

REFERENCES

[bis88] D. Barrington, N. Immerman, and H. Straubing, *On uniformity within* NC^1, in IEEE Structures in Complexity Theory, IEEE Computer Society, Washington, DC, 1988, pp. 47–59. Revised version to appear in J. Comput. System Sci.

[bo77] A. Borodin, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.

[br74] R. Brent, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[bu87] S. Buss, *The Boolean formula value problem is in* ALOGTIME, in Proc. 19th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 123–131.

[cks81] A. CHANDRA, D. KOZEN, AND L. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114–133.

[co85] S. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985) pp. 2–22.

[csv82] A. CHANDRA, L. STOCKMEYER, AND U. VISHKIN, *Complexity theory for unbounded fan-in parallelism*, in Proc. 23rd IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1982, pp. 1–13.

[dy88] P. DYMOND, *Input-driven languages are in* $\log n$ *depth*, Information Process. Lett., 26 (1988), pp. 247–250.

[dt85] P. DYMOND AND M. TOMPA, *Speedups of deterministic machines by synchronous parallel machines*, J. Comput. System Sci., 30 (1985), pp. 149–161.

[gu85] A. GUPTA, *A fast parallel algorithm for recognition of parenthesis languages*, Master's Thesis, University of Toronto, Canada, 1985.

[im89] N. IMMERMAN, *Expressibility and parallel complexity*, SIAM J. Comput., 8 (1989), pp. 625–638.

[js82] M. JERRUM AND M. SNIR, *Some exact complexity results for straight-line computations over semirings*, J. Assoc. Comput. Mach., 29 (1982), pp. 874–897.

[kr90] R. KARP AND V. RAMACHANDRAN, *Parallel algorithms for shared-memory machines*, in J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, Elsevier, New York, 1990, pp. 869–941.

[lsh65] P. LEWIS, R. STEARNS, AND J. HARTMANIS, *Memory bounds for recognition of context-free and context-sensitive languages*, in Proc. 6th Annual IEEE Symposium on Switching Circuit Theory and Logical Design, IEEE Computer Society, Washington, DC, 1965, pp. 191–202.

[ly77] N. LYNCH, *Log space recognition and translation of parenthesis languages*, J. Assoc. Comput. Mach., 24 (1977), pp. 583–590.

[mr85] G. MILLER AND J. REIF, *Parallel tree contraction and its application*, in Proc. 26th Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1985, pp. 478–489.

[mp88] D. E. MULLER AND F. PREPARATA, *Parallel restructuring and evaluation of expressions*, Tech. Report UILU-ENG-88-2253 ACT-101, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, October 1988.

[ra86] V. RAMACHANDRAN, *Restructuring formula trees*, unpublished manuscript, May 1986.

[ru81] W. RUZZO, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

[sa76] J. SAVAGE, *The Complexity of Computing*, Wiley–Interscience, Toronto, 1976.

[si83] M. SIPSER, *Borel sets and circuit complexity*, in Proc. 15th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 61–69.

[sp71] P. SPIRA, *On time hardware complexity tradeoffs for Boolean functions*, in Proc. 4th Hawaii International Symposium on System Sciences, 1971, pp. 525–527.

[to85] M. TOMPA, *A pebble game that models alternation*, unpublished manuscript, 1985.

[jg86] J. VON ZUR GATHEN, *Parallel arithmetic computations: a survey*, in Proc. 12th International Symposium on Mathematical Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1986.

SIAM J. COMPUT.
Vol. 21, No. 4, pp. 781–799, August 1992

© 1992 Society for Industrial and Applied Mathematics
010

ON THE COMPLEXITY OF POLYNOMIAL ZEROS*

DARIO BINI[†] AND LUCA GEMIGNANI[‡]

Abstract. The parallel complexity of the simultaneous approximation to all the zeros of a polynomial is investigated. By modifying and analyzing an algorithm given by Householder, it is possible to obtain a priori bounds to the number of iterations sufficient to yield a given accuracy, and to the number of digits required in the finite arithmetic. More classes of polynomials, for which the simultaneous approximation to all the zeros can be carried out in polylogarithmic time, are found. Some cases of polynomials, customarily considered hard, are easily solved. The root-finding problem for a polynomial of degree n, having zeros $z_i,\ i = 1, \cdots, n$ is $\mathcal{NC}$-reduced to finding a polynomial $a(z)$ such that $|a(z_{i+1})/a(z_i)| \leq 1 - 1/n^c$, where c is a constant.

Key words. polynomial zeros, parallel complexity, Euclidean scheme

AMS(MOS) subject classifications. 68Q25, 65H05

1. Introduction. We want to analyze the complexity of the numerical approximation of the zeros of a polynomial in a parallel model of computation. For this purpose we observe that we can restrict our investigation to the case of polynomials having integer coefficients. In fact, the case of polynomials $p(z)$ having complex coefficients can be reduced to the case of polynomials having real coefficients by considering the polynomial $p(z)\bar{p}(z)$, where $\bar{p}(z)$ is the complex conjugate of $p(z)$. The case of polynomials having real coefficients can be reduced to the case of polynomials having rational coefficients by means of rational approximation. The case of polynomials having rational coefficients can be reduced to the case of integer coefficients by means of scaling.

In this framework the complexity of the problem depends on the degree of the polynomial, on the number of digits of the coefficients, and the number of correct digits of the approximation to the zeros. Thus we can state the problem in the following way:

PROBLEM 1.1 (Polynomial root-finding). *Given positive integers n, m, and d, compute approximations to all the zeros of the polynomial $p(z) = \sum_{i=0}^{n} p_i z^i$, with absolute error at most 2^{-d}, where p_i are integers such that $|p_i| \leq 2^m,\ p_n \neq 0$.*

A wide literature exists concerning numerical algorithms for the solution of the polynomial root-finding problem. More recently, this problem has been analyzed in terms of its computational complexity [2], [17]–[22].

In a parallel-arithmetic model of computation we assume that at each step each processor can perform an arithmetic operation, and we denote by $O_A(f(n,m,d), g(n,m,d))$ the arithmetic cost of an algorithm that solves the problem in at most $kf(n,m,d)$ steps (k constant) by using $g(n,m,d)$ processors.

In a parallel Boolean model of computation we assume that at each step each processor can perform a Boolean operation, and we denote by $O_B(f(n,m,d), g(n,m,d))$ the Boolean cost of the algorithm.

We say that a problem belongs to the class $\mathcal{NC}$ in Boolean (arithmetic) sense if it can be solved with *polylogarithmic cost*, that is, with cost $O_B(f(n,m,d), g(n,m,d))$ $(O_A(f(n,m,d), g(n,m,d)))$, where f and g are polynomials in $\log n,\ \log m,\ \log d$, and in n, m, and d, respectively.

We recall that arithmetic operations modulo 2^c, as well as arithmetic operations between c-digit floating-point binary numbers, can be performed with a Boolean cost that

*Received by the editors July 12, 1989; accepted for publication (in revised form) July 24, 1991. This work was supported by Ministero dell'Università e della Ricerca Scientifica e Tecnologica 60% and 40% funds.

[†]Dipartimento di Matematica, Università di Pisa, via Buonarroti, 56100, Pisa, Italy.

[‡]Dipartimento di Matematica, Università di Parma, Via M. D'azeglio 85, 43100, Parma, Italy.

is polylogarithmic in c [13]. Therefore, an algorithm having polylogarithmic arithmetic cost also has Boolean polylogarithmic cost if the number of arithmetic digits, sufficient to give the correct output, is a polynomial function of the input sizes m, n, and d. Moreover, if the latter condition is satisfied, the algorithm can be implemented in a parallel model of computation by increasing the asymptotic number of Boolean steps by the factor $\log(nmd)$, and the number of processors by a polynomial factor.

In the case where the polynomial $p(x)$ has real zeros it can be proved [2] that Problem 1.1 belongs to $\mathcal{NC}$ in the Boolean sense. In the general case, using geometric constructions Pan [17] has shown an algorithm having a parallel cost $O_A(n \log n \log(n(d+m)), n \log(m+d)/\log(n(m+d)))$, which requires a polynomial number of digits. He proved also that for the polynomials having zeros $z_i,\ i=1,\cdots,n$ such that $|z_i/z_{i+1}| < 1 - 1/n^c$, where $|z_1| < |z_2| < \cdots < |z_n|$ and c is a constant, Problem 1.1 is in $\mathcal{NC}$ [18]. The general proof that Problem 1.1 belongs to $\mathcal{NC}$ was recently given by Neff [15].

In this paper we consider an algorithm for the root-finding problem (§2) introduced by Householder [11], and we propose and analyze a modification of it that is suitable for parallel computation (§3). The algorithm consists of two stages: in the first stage we choose a polynomial $a(z)$, a positive integer ν and we compute $a^{(\nu)}(z) = (a(z))^{\mu} \bmod p(z)$, $\mu = 2^{\nu}$; in the second stage we apply the Euclidean scheme to $p(z)$ and $a^{(\nu)}(z)$.

We prove that each term of the polynomial remainder sequence and all quotients generated in the Euclidean scheme converge to factors of $p(z)$ as $\mu \to \infty$ if

$$|a(z_1)| < |a(z_2)| < \cdots < |a(z_n)|,$$

where $z_1, \cdots, z_n$ are the zeros of $p(z)$.

The number ν of iterations needed in the first stage to approximate the result within the error bound 2^{-d}, grows polynomially with $\log n$, $\log d$, and $\log m$ if

$$\left|\frac{a(z_j)}{a(z_{j+1})}\right| < 1 - \frac{1}{n^c}, \quad j = 1, \cdots, n-1, \tag{1.1}$$

where c is a constant. Moreover, if the inequality in (1.1) holds only for $j = k$, then the algorithm delivers approximations within the error bound 2^{-d} of just two factors $p_k(z)$ and $p_{n-k}(z)$ of degree k and $n-k$ such that $p(z) = p_k(z)p_{n-k}(z)$.

We prove that the cost of the algorithm is given by $O_A(\nu \log n, n \log\log n) + O_A(\log^2 n, n^3)$ or, alternatively, $O_A(\nu \log n, n \log\log n) + O_A(\log^3 n, n^2)$ if the Euclidean scheme computation of [2] is applied.

The number of binary digits of the floating-point arithmetic, sufficient to compute the result with the given precision, depends polynomially on the input sizes d, n, and m. That is, the algorithm can be carried out with a polylogarithmic Boolean cost.

In §4 we discuss a suitable strategy for choosing the polynomial $a(z)$ that guarantees that the inequality (1.1) is satisfied for at least one j. This leads to a polylogarithmic algorithm for the computation of a single zero. For the computation of all the zeros the worst-case arithmetic time-cost is the same as that of Pan's algorithm. Our algorithm generalizes the result of [18] (obtained as a particular case with $a(z) = z$), since it performs the computation with a polylogarithmic cost whenever (1.1) holds.

Some cases of clustered zeros that are customarily considered hard are easily solved in §4. For instance, we prove that the case where the clustered zeros are roughly in geometric progression (i.e., $z_1 = \eta$, $z_2 = \eta + \epsilon\gamma_1$, $z_3 = \eta + \epsilon\gamma_1 + \epsilon^2\gamma_2$, and $z_4 = \eta + \epsilon\gamma_1 + \epsilon^2\gamma_2 + \epsilon^3\gamma_3, \cdots$, where ϵ is a small positive number and $\gamma_i,\ i = 1, \cdots, n-1$, are complex numbers having almost the same moduli) can be solved with polylogarithmic cost by applying the algorithm with $a(z) = p'(z)$.

In the case where the polynomial $p(z)$ has real zeros, once a separator w of the zeros has been computed as in [2], our algorithm can be used to factor $p(z)$ as a product of two polynomials having degrees lower than $3n/4$, without performing contour integrations. This way, all the zeros can be recursively approximated with polylogarithmic cost.

2. Householder's algorithm. Here we present a slightly simpler version of an algorithm given by Householder in [11]. Let $p(z)$ and $a(z)$ be polynomials with complex coefficients such that

$$p(z) = \prod_{i=1}^{n}(z - z_i), \quad z_i \neq 0, \quad i = 1, \cdots, n,$$

and degree $(a(z)) < n$. Hereafter, we set $||p(z)||_\infty = \max_{0 \le i \le n} |p_i|$. Suppose that the zeros z_i of $p(z)$ have been ordered so that $|\lambda_i|/|\lambda_{i+1}| \le 1$, $i = 1, \cdots, n$, where $\lambda_i = a(z_i)$. We denote by $p_{i_1, i_2, \cdots, i_k}(z)$ the polynomial $p(z)/((z - z_{i_1})(z - z_{i_2}) \cdots (z - z_{i_k}))$.

Consider the following algorithm:

ALGORITHM 2.1.

1. Given an integer ν, compute $a^{(\nu)}(z) = a(z)^{2^\nu} \bmod p(z)$, in the following way:

$$a^{(0)}(z) = a(z),$$
$$a^{(i+1)}(z) = h^{(i)}(z)^2 \bmod p(z), \qquad i = 0, \cdots, \nu - 1.$$

2. Apply the Euclidean algorithm to the pair of monic polynomials $u_n(z) = p(z)$, $u_{n-1}(z) = a^{(\nu)}(z)/\beta_0$, where β_0 is the leading coefficient of $a^{(\nu)}$, obtaining

$$u_{n-i+1}(z) = u_{n-i}(z)(z - \alpha_i) - \beta_i u_{n-i-1}(z), \qquad i = 1, \cdots, n-1,$$
$$u_1(z) = z - \alpha_n, \; u_0(z) = 1,$$

where we suppose that $\beta_i \neq 0$, $i = 0, \cdots, n-1$, and $u_{n-i}(z)$ is a monic polynomial of degree $n - i$, $i = 1, \cdots, n$.

Here the quantities α_i, β_i, u_{n-i} depend also on ν, i.e., $\alpha_i = \alpha_i^{(\nu)}$, $\beta_i = \beta_i^{(\nu)}$, $u_{n-i} = u_{n-i}^{(\nu)}$. For the sake of simplicity we omit the superscript if it is not strictly needed.

It is easy to prove that, for almost any polynomial $a(z)$, Algorithm 2.1 can be carried out. We have, in fact, the following result.

PROPOSITION 2.1. *Let Γ_ν be the set of vectors $(a_0, \cdots, a_{n-1}) \in \mathbf{C}^n$ such that Algorithm 2.1, with $a(z) = \sum_{i=0}^{n-1} a_i z^i$, there exists an integer j, $0 \le j \le n-1$, such that $\beta_j^{(\nu)} = 0$. Then the set $\bigcup_{\nu \in \mathbf{N}} \Gamma_\nu$ has zero Lebesgue measure over $\mathbf{C}^n$. In other words, for almost any polynomial $a(z)$, stage 2 of Algorithm 2.1 can be carried out if $\beta_i^{(\nu)} \neq 0$, $i = 0, \cdots, n-1$, and* degree$(u_{n-i}^{(\nu)}(z)) = n - i$, $i = 1, \cdots, n-1$.

Proof. By construction, $\beta_i^{(\nu)}$ is a rational function of the variables $a_0, \cdots, a_{n-1}$, $i = 0, \cdots, n-1$. Since the set of the zeros of a (identically nonzero) rational function from $\mathbf{C}^n$ to $\mathbf{C}$ is a zero-measure subset of $\mathbf{C}^n$, we have that Γ_ν has zero measure over $\mathbf{C}^n$; therefore, $\bigcup_{\nu \in \mathbf{N}} \Gamma_\nu$ also has zero measure over $\mathbf{C}^n$. □

We observe that stage 1 can be performed in $O(\nu n \log n)$ arithmetic operations by using fast polynomial arithmetic [5], and stage 2, i.e., the computation of α_i, β_{i-1}, $i = 1, \cdots, n$, can be carried out in $O(n \log^2 n)$ arithmetic operations by means of the extended Euclidean algorithm [1]. However, it is worth pointing out that the evaluation of all the coefficients of the polynomials $u_i(z)$, $i = 1, \cdots, n$, would cost at least $n^2/2$

arithmetic operations, whereas the computation of $u_j(z)$ for a given j (together with α_i, β_{i-1}, $i=1,\cdots,n$) still costs $O(n\log^2 n)$ arithmetic operations [7].

In a parallel (algebraic) model of computation, the cost of stage 1 is $O_A(\nu\log n, n\log\log n)$ (compare [4]) and the cost of stage 2 is $O_A(\log^3 n, n^2)$. In fact, the Euclidean algorithm can be reduced to computing Sylvester-like determinants [10] (see also §3), and this task can be solved with cost $O_A(\log^2 n, n^2)$ [19]. Alternatively, it can be reduced to computing the LU block triangular factorization of an $n\times n$ Hankel matrix generated by the rational function $u_{n-1}(z)/u_n(z)$ and this task can be solved with cost $O_A(\log^3 n, n^2)$ [3].

Algorithm 2.1 can be used for the simultaneous approximation to the zeros of the polynomial $p(z)$. In fact, if $p(z)$ has pairwise different zeros, it is possible to prove (see [10] and Proposition 3.1) that

$$||u_{n-i}(z)-p_{1,2,\cdots,i}(z)||_\infty = O\left(\left|\frac{\lambda_j}{\lambda_{j+1}}\right|^{2^\nu}\right),$$

$$|\alpha_j - z_j| = O\left(\left|\frac{\lambda_{j+1}}{\lambda_j}\right|^{2^\nu} + \left|\frac{\lambda_j}{\lambda_{j-1}}\right|^{2^\nu}\right)\quad j=2,\cdots,n-1,$$

$$|\alpha_1 - z_1| = O\left(\left|\frac{\lambda_2}{\lambda_1}\right|^{2^\nu}\right),\quad |\alpha_n - z_n| = O\left(\left|\frac{\lambda_n}{\lambda_{n-1}}\right|^{2^\nu}\right),$$

$$|\beta_j| = O\left(\left|\frac{\lambda_{j+1}}{\lambda_j}\right|^{2^\nu}\right).$$

That is, the algorithm has a global quadratic convergence. Observe that the condition of distinct zeros is not restrictive because it is possible to deflate the multiple zeros, computing the $\gcd(p,p')$ by means of the Euclidean algorithm (compare [9] and [3]).

This result suggests that Algorithm 2.1 be used in the following way. Choose a random polynomial $a(z)$ and compute, for a given ν, the coefficients α_i and β_i. Select the subscript i, closest to $n/2$, such that $|\beta_i|$ is small enough, and compute the coefficients of $u_{n-i}(z)$. Then compute the quotient $q(z)$ of the division between $p(z)$ and $u_{n-i}(z)$ and recursively apply this algorithm to $q(z)$ and $u_{n-i}(z)$ until linear polynomials are obtained.

3. Parallel implementation and convergence analysis. In order to give explicit a priori bounds to the number of digits of the arithmetic and to the number of iterations needed to reach a given accuracy, we now describe a different implementation of Householder's algorithm. First, observe that stage 2 of Algorithm 2.1 yields the recurrence

$$\tilde{u}_{n-i+1}(z) = (1-z\alpha_i)\tilde{u}_{n-i}(z) - z^2\beta_i\tilde{u}_{n-i-1}(z),$$

where $\tilde{u}_j(z) = z^j u_j(z^{-1})$. Whence we have

$$\frac{\tilde{u}_{n-i+1}(z)}{\tilde{u}_{n-i}(z)} = 1 - z\alpha_i - \frac{z^2\beta_i}{\dfrac{\tilde{u}_{n-i}(z)}{\tilde{u}_{n-i-1}(z)}},$$

which gives the continued fraction decomposition of the function $g(z) = \frac{a^{(\nu)}(z^{-1})}{zp(z^{-1})}$, i.e.,

$$g(z) = \cfrac{\beta_0}{1 - z\alpha_1 - \cfrac{z^2\beta_1}{1 - z\alpha_2 - \cfrac{z^2\beta_2}{1 - z\alpha_3 + \cdots}}}.$$

From the theory of Padé approximants we have the relations ([10], pp. 60–64)

$$\begin{aligned} &\alpha_i = q_i + e_{i-1}, \qquad \beta_i = q_i e_i, \qquad i = 1, \cdots, n, \qquad (e_0 = 0,\ e_n = 0), \\ &q_i = \frac{h_{i,i}h_{i-2,i-1}}{h_{i-1,i}h_{i-1,i-1}}, \qquad e_i = \frac{h_{i,i+1}h_{i-1,i-1}}{h_{i-1,i}h_{i,i}}, \end{aligned} \tag{3.1}$$

where $h_{i,j}$, is the Hankel determinant obtained from the coefficients of the power series

$$g(z) = \frac{a^{(\nu)}(z^{-1})}{zp(z^{-1})} = \sum_{i=0}^{+\infty} c_i z^i,$$

that is,

$$h_{i,j} = \det H_{i,j}, \quad H_{i,j} = (c_{r+s+i-j-1})_{r,s=1,j}.$$

(Here we assume $h_{i,0} = 1$, $h_{i,j} = c_i = 0$ if $i < 0$, $j \neq 0$.) Moreover, if $a^{(\nu)}(z) = \sum_{i=0}^{n-1} a_i^{(\nu)} z^i$, for the Hankel determinants $h_{i,j}$ we have the following formula ([10], p. 55):

$$\begin{aligned} &h_{i,j} = a_{n-1}^{-i-j}(-1)^j \det R_{i,j}, \\ &R_{i,j} = \left(\begin{array}{cccc} p_n & p_{n-1} & p_{n-2} & \cdots \\ & p_n & p_{n-1} & \cdots \\ \cdots & \cdots & \cdots & \\ a_{n-1}^{(\nu)} & a_{n-2}^{(\nu)} & a_{n-3}^{(\nu)} & \cdots \\ & a_{n-1}^{(\nu)} & a_{n-2}^{(\nu)} & \cdots \\ \cdots & \cdots & \cdots & \end{array} \right) \begin{array}{l} \left.\vphantom{\begin{array}{c} a \\ a \\ a \end{array}}\right\} j \\ \left.\vphantom{\begin{array}{c} a \\ a \\ a \end{array}}\right\} j \end{array}. \end{aligned} \tag{3.2}$$

Let $A_{i,j}(z)$ be the matrix obtained by replacing the last column of $H_{i+1,j+1}$ with the vector $(z^j, z^{j-1}, \cdots, 1)^T$, that is,

$$A_{i,j}(z) = \left((c_{r+s+i-j-1})_{\substack{r=1,j+1 \\ s=1,j}} \ \middle|\ \begin{pmatrix} z^j \\ \vdots \\ 1 \end{pmatrix} \right).$$

For the Hankel polynomial defined by $k_{i,j}(z) = \det A_{i,j}(z)/h_{i,j}$, the following recurrence holds ([10], p. 64):

$$k_{j,j+1}(z) = (1 - z\alpha_j)k_{j-1,j}(z) - \beta_j z^2 k_{j-2,j-1}(z).$$

That is, the reversed polynomial $t_j(z) = z^j k_{j-1,j}(z^{-1})$ satisfies the following relation:

$$t_{j+1}(z) = (z - \alpha_j)t_j(z) - \beta_j t_{j-1}(z).$$

Moreover, applying the Laplace rule to compute $\det(zI - T_{j+1})$, where

$$T_{j+1} = \begin{pmatrix} \alpha_1 & \beta_1 & & & & \\ 1 & \alpha_2 & \beta_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & \alpha_j & \beta_j \\ & & & & 1 & \alpha_{j+1} \end{pmatrix},$$

yields the recurrence

$$\det(zI - T_{j+1}) = (z - \alpha_j)\det(zI - T_j) - \beta_j \det(zI - T_{j-1}).$$

Therefore, since $t_1(z) = z - \alpha_1 = \det(zI - T_1)$ and $t_0(z) = 0$, by using the induction argument we obtain

$$t_j(z) = z^j k_{j-1,j}(z^{-1}) = \det(zI - T_j). \tag{3.3}$$

Now we are ready to provide a different version of Algorithm 2.1 that is particularly suitable for a parallel implementation.

ALGORITHM 3.1.

1. Compute the coefficients of the polynomial $a^{(\nu)}(z) = a(z)^\mu \bmod p(z)$, $\mu = 2^\nu$ by means of repeated squaring modulo $p(z)$.
2. Compute $h_{i,j}$ according to formula (3.2).
3. Compute α_i, β_i, according to formulae (3.1).
4. Compute the coefficients of the polynomial $t_i(z) = \det(zI - T_i)$, $i = 1, \cdots, n-1$.

Stage 1 costs $O_A(\nu \log n, n \log\log n)$, and stage 2 costs $O_A(\log^2 n, n^3)$ because the evaluation of the determinant of a "quasi Toeplitz" matrix can be performed with cost $O_A(\log^2 n, n^2)$ [19]. Stage 4 can be performed with cost $O_A(\log^2 n, n^2)$ by applying the parallel prefix scheme [16] for the computation of $\prod_{j=1}^{i} B_j$, $i = 1, \cdots, n-1$, where $B_i = \begin{pmatrix} z - \alpha_i & -\beta_i \\ 1 & 0 \end{pmatrix}$. In fact,

$$\begin{pmatrix} t_{i+1} \\ t_i \end{pmatrix} = \prod_{j=1}^{i} B_j \begin{pmatrix} t_1 \\ t_0 \end{pmatrix}.$$

Therefore, the overall cost of Algorithm 3.1 is $O_A(\nu \log n, n \log\log n) + O_A(\log^2 n, n^3)$.

The following proposition strengthens the convergence results of [11], yielding bounds to the number ν of iterations sufficient to obtain a given accuracy.

PROPOSITION 3.1. *Let $p(z) = \prod_{i=1,n}(z - z_i)$ be a polynomial having pairwise different zeros z_i, $i = 1, \cdots, n$ ordered so that $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|$, where $\lambda_i = a(z_i)$ and $a(z)$ is a polynomial of degree less than n such that Algorithm 3.1 can be carried out. Set $z_{\max} = \max|z_i|$, $z_{\min} = \min|z_i| \neq 0$, and $\Delta = \min_{i\neq j}|z_i - z_j|$. Then we have*

$$\beta_j = \left(\frac{\lambda_{j+1}}{\lambda_j}\right)^\mu B_j \frac{(1 + \theta_{0,j-1})(1 + \theta_{0,j+1})}{(1 + \theta_{0,j})^2}, \qquad j = 1, 2, \cdots, n-1,$$

$$\alpha_j - z_j = A_j^{(1)}\left(\frac{\lambda_{j+1}}{\lambda_j}\right)^\mu + \left(\frac{\lambda_j}{\lambda_{j-1}}\right)^\mu A_j^{(2)}, \qquad j = 1, 2, \cdots, n,$$

where

$$|B_j| \le \frac{(2z_{\max})^n}{\Delta^{n-2}},$$

$$\theta_{i,j} = \left(\frac{\lambda_{j+1}}{\lambda_j}\right)^{\mu} \varphi_{i,j}, \; |\varphi_{i,j}| \le \left(\binom{n}{j} - 1\right) \left|\frac{z_{\max}}{z_{\min}}\right|^{ij} \left(\frac{2z_{\max}}{\Delta}\right)^{n-j},$$

$$|A_j^{(1)}| \le \frac{z_{\max}|\varphi_{1,j}|(2+|\theta_{0,j-1}|)(1+|\theta_{1,j-1}|)}{|(1+\theta_{1,j-1})(1+\theta_{0,j})|},$$

$$|A_j^{(2)}| \le \frac{z_{\max}(|\varphi_{0,j-1}|+|\varphi_{1,j-1}|)}{|(1+\theta_{1,j-1})(1+\theta_{0,j})|} + \frac{|B_{j-1}|}{z_{\min}}\left|\frac{(1+\theta_{0,j})(1+\theta_{1,j-2})}{(1+\theta_{1,j-1})(1+\theta_{0,j})}\right|.$$

Proof. First we express $h_{i,j}$ in terms of the zeros z_i, $i = 1, \cdots, n$, of $p(z)$. For this purpose we prove that

$$c_m = \sum_{i=1}^{n} z_i^m S_i, \qquad m = 1, 2, \cdots, \qquad S_i = a^{(\nu)}(z_i)/p'(z_i). \tag{3.4}$$

In fact, from the Lagrange representation of $a^{(\nu)}(z)$ with respect to $z_1, \cdots, z_n$, we have

$$g(z) = \sum_{i=1}^{n} \frac{a^{(\nu)}(z_i)}{p'(z_i)} \frac{1}{1 - zz_i}.$$

Therefore,

$$c_m = \frac{1}{m!} D^m g(z)\Big|_{z=0} = \frac{1}{m!} \sum_{i=1}^{n} \frac{a^{(\nu)}(z_i)}{p'(z_i)} D^m \frac{1}{1-zz_i}\Big|_{z=0} = \sum_{i=1}^{n} \frac{a^{(\nu)}(z_i)}{p'(z_i)} z_i^m.$$

Then, from the properties of determinants, we have

$$h_{i,j} = \det(c_{r+s+i-j-1})_{r,s=1,j} = \sum_{\omega\in\Omega} \det(z_{\omega(s)}^{r+s+i-j-1} S_{\omega(s)})_{r,s=1,j},$$

where $\Omega = \{\omega : \{1, 2, \cdots, j\} \to \{1, 2, \cdots, n\}\}$ is the set of all the functions from $\{1, 2, \cdots, j\}$ to $\{1, 2, \cdots, n\}$. Therefore,

$$h_{i,j} = \sum_{\omega\in\Omega} \prod_{k=1}^{j} (S_{\omega(k)} z_{\omega(k)}^{i-j+1}) \det(z_{\omega(s)}^{r+s-2})_{r,s=1,j}.$$

Now, it is easy to check that the matrix $(z_{\omega(s)}^{r+s-2})_{r,s=1,j}$ is singular if $\omega(s_1) = \omega(s_2)$ for $s_1 \ne s_2$. (In fact, in this case the columns with subscript s_1 and s_2, respectively, are linearly dependent.) Therefore, we can assume that Ω is the set of all the injective functions from $\{1, 2, \cdots, j\}$ to $\{1, 2, \cdots, n\}$ so that partitioning Ω as $\Omega = \bigcup_{q=1}^{\binom{n}{j}} \Omega_q$, where Ω_q, $q = 1, \cdots, \binom{n}{j}$ are the equivalence classes defined by the equivalence relation

$$\omega_1 \sim \omega_2 \quad \text{if} \quad \omega_1(\{1, 2, \cdots, j\}) = \omega_2(\{1, 2, \cdots, j\}),$$

we obtain

$$h_{i,j} = \sum_{q=1}^{\binom{n}{j}} \left(\prod_{k=1}^{j} S_{\omega_q(k)} z_{\omega_q(k)}^{i-j+1} \right) \sum_{\omega\in\Omega_q} \det(z_{\omega(s)}^{r+s-2})_{r,s=1,j}, \tag{3.5}$$

where ω_q is any function in Ω_q. Now, since

$$\sum_{\omega\in\Omega_q} \det(z_{\omega(s)}^{r+s-2})_{r,s=1,j} = \prod_{\omega_q(r)<\omega_q(s)} (z_{\omega_q(r)} - z_{\omega_q(s)})^2,$$

for any $\omega_q \in \Omega$ (compare [12]) we have

$$h_{i,j} = \prod_{k=1}^{j} S_k z_k^{i-j+1} \prod_{1\le r<s\le} (z_r - z_s)^2 + \sum_{q=2}^{\binom{n}{j}} \left(\prod_{k=1}^{j} S_{\omega_q(k)} z_{\omega_q(k)}^{i-j+1} \prod_{\omega_q(r)<\omega_q(s)} (z_{\omega_q(r)} - z_{\omega_q(s)})^2 \right),$$

where we assume $\omega_1(r) = r$, $r = 1, \cdots, j$. Therefore, we obtain

$$h_{i,j} = \prod_{k=1}^{j} S_k z_k^{i-j+1} \prod_{1\le r<s\le j} (z_r - z_s)^2 (1 + \theta_{i-j+1,j}), \tag{3.6}$$

$$\theta_{i-j+1,j} = \frac{\sum_{q=2}^{\binom{n}{j}} \left(\prod_{k=1}^{j} S_{\omega_q(k)} z_{\omega_q(k)}^{i-j+1} \prod_{\omega_q(r)<\omega_q(s)} (z_{\omega_q(r)} - z_{\omega_q(s)})^2 \right)}{\prod_{k=1}^{j} s_k z_k^{i-j+1} \prod_{1\le r<s\le j} (z_r - z_s)^2}. \tag{3.7}$$

Observe that

$$\prod_{k=1}^{j} S_{\omega_q(k)} z_{\omega_q(k)}^{i-j+1} \prod_{\omega_q(r)<\omega_q(s)} (z_{\omega_q(r)} - z_{\omega_q(s)})^2 = - \prod_{k=1}^{j} \frac{a^{(\nu)}(z_{\omega_q(k)}) z_{\omega_q(k)}^{i-j+1}}{p_{\omega_q(1),\omega_q(2),\cdots,\omega_q(j)}(z_{\omega_q(k)})}.$$

Moreover,

$$\left| \prod_{k=1}^{j} \frac{a^{(\nu)}(z_{\omega_q(k)})}{a^{(\nu)}(z_k)} \right| = \prod_{k=1}^{j} \left| \frac{\lambda_{\omega_q(k)}}{\lambda_k} \right|^{2^\nu} \le \left| \frac{\lambda_{j+1}}{\lambda_j} \right|^{2^\nu},$$

so that

$$\begin{aligned} \theta_{i-j+1,j} &= \left(\frac{\lambda_{j+1}}{\lambda_j} \right)^{2^\nu} \varphi_{i-j+1,j} \\ |\varphi_{i-j+1,j}| &\le \left(\binom{n}{j} - 1 \right) \max_{\omega\in\Omega} \left(\prod_{k=1}^{j} \left| \frac{z_{\omega(k)}}{z_k} \right|^{i-j+1} \left| \frac{p_{1,2,\cdots,j}(z_k)}{p_{\omega(1),\omega(2),\cdots,\omega(j)}(z_{\omega(k)})} \right| \right). \end{aligned} \tag{3.8}$$

Now we are ready to obtain the bounds on $|\beta_j|$ and on $|\alpha_j - z_j|$. In fact, from (3.1), we obtain

$$\beta_j = q_j e_j = \frac{h_{j-2,j-1} h_{j,j+1}}{h_{j-1,j}^2},$$

and from (3.6) we have

$$\beta_j = \frac{S_{j+1}}{S_j} \frac{\prod_{r=1}^{j}(z_r - z_{j+1})^2}{\prod_{r=1}^{j-1}(z_r - z_j)^2} \frac{(1+\theta_{0,j-1})(1+\theta_{0,j+1})}{(1+\theta_{0,j})^2}.$$

Therefore, using (3.4) we obtain

$$\beta_j = \left(\frac{\lambda_{j+1}}{\lambda_j}\right)^{2^\nu} B_j \frac{(1+\theta_{0,j-1})(1+\theta_{0,j+1})}{(1+\theta_{0,j})^2}$$

$$|B_j| = \left|\frac{p_{1,2,\cdots,j}(z_j) p_{j+1,j+2,\cdots,n}(z_{j+1})}{p_{1,2,\cdots,j+1}(z_{j+1}) p_{j,j+1,\cdots,n}(z_j)}\right|.$$

Concerning $|\alpha_i - z_i|$, since from (3.1) we have $\alpha_j = q_j + e_{j-1} = q_j + \beta_{j-1}/q_{j-1}$ and from (3.7) we have

$$q_j = z_j \frac{(1+\theta_{1,j})(1+\theta_{0,j-1})}{(1+\theta_{1,j-1})(1+\theta_{0,j})},$$

we obtain

$$(\alpha_j - z_j) = z_j \left(\frac{(1+\theta_{1,j})(1+\theta_{0,j-1})}{(1+\theta_{1,j-1})(1+\theta_{0,j})} - 1\right) + \frac{\beta_{j-1}}{z_{j-1}} \frac{(1+\theta_{1,j-2})(1+\theta_{0,j-1})}{(1+\theta_{1,j-1})(1+\theta_{0,j-2})}, \; j = 2, \cdots, n,$$

whence

$$\alpha_j - z_j = A_j^{(1)} \left(\frac{\lambda_{j+1}}{\lambda_j}\right)^{2^\nu} + \left(\frac{\lambda_j}{\lambda_{j-1}}\right)^{2^\nu} A_j^{(2)}, \; j = 1, \cdots, n,$$

where

$$A_j^{(1)} = \frac{z_j((1+\theta_{0,j-1})\varphi_{1,j} - \varphi_{0,j})(1+\theta_{1,j-1})}{(1+\theta_{1,j-1})(1+\theta_{0,j})}, \qquad j = 1, \cdots, n.$$

$$A_j^{(2)} = \frac{z_j(\varphi_{0,j-1} - \varphi_{1,j-1})}{(1+\theta_{1,j-1})(1+\theta_{0,j})} + \frac{B_{j-1}}{z_{j-1}} \frac{(1+\theta_{1,j-2})(1+\theta_{0,j})}{(1+\theta_{1,j-1})(1+\theta_{0,j-1})}, \; j = 1, \cdots, n.$$

Here we assume $B_0 = 0$, $\varphi_{i,0} = \varphi_{1,0} = 0$. □

Consider the polynomial $\hat{t}_j(z) = \det(zI - \hat{T}_j)$, where $\hat{T}_j$ is the $j \times j$ principal submatrix of T_n made up by the last j rows and columns, then, by using (3.3), we can prove the following.

PROPOSITION 3.2. *We have $u_i(z) = \hat{t}_i(z)$, $i = 0, \cdots, n$, and, in particular, the zeros of $p(z)$ are the eigenvalues of the matrix T_n. Moreover, under the assumption and the notations of Proposition 3.1, we have that for almost any polynomial $a(z)$ the following relation holds:*

$$||t_j(z) - p_{j+1,\cdots,n}(z)||_\infty \le \left|\frac{\lambda_j}{\lambda_{j+1}}\right|^{2^\nu} \left|\frac{\varphi_{0,j}}{1+\theta_{0,j}}\right| D_j,$$

where

$$D_j = \max_{\omega\in\Omega}\left\|\prod_{r=1}^{j}(z-z_r)-\prod_{r=1}^{j}(z-z_{\omega(r)})\right\|_\infty .$$

Moreover,

$$||u_{n-j}(z)-p_{1,2,\cdots,j}(z)||_\infty = O\left(\left|\frac{\lambda_j}{\lambda_{j+1}}\right|^{2^\nu}\right).$$

Proof The relation $u_i(z)=\hat{t}_i(z),\ i=0,\cdots,n$ can be easily proved through induction by applying the Laplace rule for determinants since $u_0(z)=\hat{t}_0(z)=1,\ u_1(z)=\hat{t}_1(z)=z-\alpha_n$. For the second part, applying to $k_{j-1,j}(z)$ the same arguments used in the proof of Proposition 3.1 yields

$$h_{j-1,j}t_j(z)=z^jk_{j-1,j}(z^{-1})=\sum_{q=1}^{\binom{n}{j}}\left(\prod_{k=1}^{j}S_{\omega_q(k)}\sum_{\omega\in\Omega_q}\det\left((z_{\omega(s)}^{r+s-2})_{\substack{r=1,j+1\\ s=1,j}}\middle|\begin{pmatrix}1\\ \vdots\\ z^j\end{pmatrix}\right)\right);$$

compare (3.3) and (3.5). Now, since the determinant in the preceding summation is a polynomial in z of degree j that is zero for $z=z_{\omega(s)},\ s=1,2,\cdots,j$, we have

$$\det\left((z_{\omega(s)}^{r+s-2})_{\substack{r=1,j+1\\ s=1,j}}\middle|\begin{pmatrix}1\\ \vdots\\ z^j\end{pmatrix}\right)=\prod_{k=1}^{j}(z-z_{\omega(k)})\det(z_{\omega(s)}^{r+s-2})_{r,s=1,j}.$$

Therefore, applying the arguments used in Proposition 3.1 yields

$$h_{j-1,j}t_j(z)=\sum_{q=1}^{\binom{n}{j}}\prod_{k=1}^{j}S_{\omega_q(k)}(z-z_{\omega_q(k)})\sum_{\omega\in\Omega_q}\det(z_{\omega(s)}^{r+s-2})_{r,s=1,j}=$$

$$\prod_{k=1}^{j}S_k(z-z_k)\prod_{1\le r<s\le j}(z_r-z_s)^2+\sum_{q=2}^{\binom{n}{j}}\prod_{k=1}^{j}s_{\omega_q(k)}(z-z_{\omega_q(k)})\prod_{\omega_q(r)<\omega_q(s)}(z_{\omega_q(r)}-z_{\omega_q(s)})^2,$$

whence

$$h_{j-1,j}t_j(z)-h_{j-1,j}p_{j+1,\cdots,n}(z)=$$

$$\sum_{q=2}^{\binom{n}{j}}\prod_{\omega_q(r)<\omega_q(s)}(z_{\omega_q(r)}-z_{\omega_q(s)})^2\prod_{k=1}^{j}s_{\omega_q(k)}\left(\prod_{r=1}^{j}(z-z_{\omega_q(r)})-\prod_{r=1}^{j}(z-z_r)\right).$$

That is, by using (3.6) and (3.7) we obtain

$$||t_j(z)-p_{j+1,\cdots,n}(z)||_\infty\le D_j\left|\frac{\lambda_{j+1}}{\lambda_j}\right|^{2^\nu}\left|\frac{\varphi_{0,j}}{1+\theta_{0,j}}\right|,$$

where

$$D_j = \max_{\omega \in \Omega} \left\| \prod_{r=1}^{j} (z - z_{\omega(r)}) - \prod_{r=1}^{j} (z - z_r) \right\|_\infty$$

and $\varphi_{0,j}$ and $\theta_{0,j}$ are defined in (3.8). The quadratic convergence of the polynomials $u_{n-j}(z)$ follows from the relation

$$p(z) = u_{n-i}(z)t_i(z) - \beta_i u_{n-i-1}(z)t_{i-1}(z),\ i = 1, 2, \cdots, n-1, \quad (3.9)$$

which is obtained by expanding $\det(zI - T_n)$ along the ith column. □

We observe that the matrix T_n is similar to the complex symmetric tridiagonal matrix having diagonal entries α_i and superdiagonal entries $\beta_i^{1/2}$. Applying Gerschgorin's theorem yields $u_i \in \bigcup_{j=1,n} C_j$ where C_j, is defined by

$$C_j = \{z \in \mathbf{C} :\ |z - \alpha_j| \le |\beta_j|^{1/2} + |\beta_{j-1}|^{1/2},\ j = 1, \cdots, n\}$$

(assuming $\beta_0 = \beta_n = 0$). Moreover, if C_j is disjoint from the other disks, then $|\alpha_j - u_j| \le |\beta_j|^{1/2} + |\beta_{j-1}|^{1/2}$.

It is interesting to note that the relations and the bounds given in Propositions 2.1 and 3.1 still hold if the polynomial $a(z)$ is such that $|\lambda_j| = |\lambda_{j+1}| = \cdots = |\lambda_{j+k}|$ for some j and k. In this case we have no convergence for the terms $\alpha_i^{(\nu)}$, $i = j, j+1, \cdots, j+k$, but the speed of convergence of the other components $\alpha_i^{(\nu)}$ is not affected by this situation. Moreover, it is sufficient to choose $a(z)$ in such a way that there is at least one break-point, i.e., an integer j for which $|\lambda_j/\lambda_{j+1}| < 1$, in order that two factors of $p(z)$ can be easily approximated. Different strategies for choosing $a(z)$ are discussed in §4.

An interesting situation occurs if $p(z)$ and $a(z)$ have real coefficients, because in this case the quantities λ_j may occur in conjugate pairs and the algorithm delivers (after performing an additional division) the quadratic factors of $p(z)$. Therefore, there is no need to perform the computation with complex arithmetic. The possibility of computing factors of $p(z)$ yields a tool that allows us to handle the case of clustered zeros.

The above results allow us to give a priori upper bounds to the number of iterations needed to approximate a zero of $p(z)$ within a given accuracy. We say that there is a *strong break-point* at j if

$$\left| \frac{\lambda_{j+1}}{\lambda_j} \right| < 1 - 1/n^c, \quad (3.10)$$

where c is a positive constant. In the next section we prove that for any polynomial $p(z)$ it is possible to compute, in polylogarithmic time, a polynomial $a(z)$ such that (3.10) is satisfied with $c = 2$.

The following properties allow us to prove interesting corollaries of Propositions 3.1 and 3.2.

PROPERTY 3.1 ([13], [14]). *If the polynomial $p(x)$ has pairwise different zeros $z_1, \cdots, z_n$, and integer coefficients p_i, $i = 0, \cdots, n$ such that $|p_i| \le 2^m$, then*

$$\Delta = \min_{i \ne j} |z_i - z_j| > 2^{-\delta}, \qquad \delta = 2n \log n + mn.$$

PROPERTY 3.2 ([10]). *For the zeros $z_1, \cdots, z_n$ of the polynomial defined in Property 3.1, the following relation holds:*

$$(2^m + 1)^{-1} \le |z_i| \le 2^m + 1.$$

COROLLARY 3.1. *Let $p(z)$ be a polynomial satisfying the conditions of Property* 3.1. *Suppose that for the polynomials $p(z)$ and $a(z)$ there is a strong break-point at j. Choosing ν in Algorithm* 3.1 *so that $\nu > \log \sigma(m,n)$, where $\sigma(m,n) = 2n^{c+2}\log(n2^m)$ yields the bound $|\theta_{0,j}| < 1/2$. Moreover, $\nu > \log(\sigma(m,n) + n^c(1+d)\log 2)$ iterations of Algorithm* 3.1 *are sufficient to compute the coefficients of the factor $p_{j+1,\cdots,n}$ with absolute error less than 2^{-d}. If there is a strong break-point at $j+1$, an analogous bound to the number of iterations holds for the approximation to the zero z_j.*

Proof. The condition $|\theta_{0,j}| < 1/2$ is implied by

$$\left|\frac{\lambda_{j+1}}{\lambda_j}\right|^{2^\nu} < \frac{1}{2}\left|\frac{\Delta}{2z_{\max}}\right|^{n-j}\frac{1}{\binom{n}{j}}.$$

Therefore, since $\binom{n}{j} \le n^j$, and $\log|\lambda_{j+1}/\lambda_j| \le \log(1-1/n^c) \le -1/n^c$, Properties 3.1 and 3.2 imply that the above condition is satisfied if

$$2^\nu \ge n^c((n-j)\log\frac{2z_{\max}}{\Delta} + j\log n + \log 2),$$

that is, if

$$2^\nu \ge n^c(2n^2\log n + n^2 m).$$

Thus we obtain the first bound. Concerning the second bound, observe that Proposition 3.2 and the inequalities $\nu \ge \log\sigma(m,n)$, $|\theta_{0,j}| < 1/2$, imply that

$$||t_j(z) - p_{j+1,\cdots,n}(z)||_\infty \le \left|1 - \frac{1}{n^c}\right| |2\varphi_{0,j}D_j|.$$

Therefore the condition

$$||t_j(z) - p_{j+1,\cdots,n}(z)||_\infty < 2^{-d}$$

is verified if

$$2^\nu \ge n^c(\log(2\varphi_{0,j}) + \log|D_j| + d\log 2).$$

The result follows from Properties 3.1 and 3.2 because

$$D_j = \max_{\omega\in\Omega}\left\|\prod_{r=1}^{j}(z-z_r) - \prod_{r=1}^{j}(z-z_{\omega(r)})\right\|_\infty \le 2\max_k\binom{j}{k}z_{\max}^k \le (1+z_{\max})^j. \qquad \square$$

COROLLARY 3.2. *Let c and $a(z)$ be given, and consider the class $\mathcal{P}_{c,a}$ of polynomials having zeros z_j such that $|a(z_{j+1})/a(z_j)| \le 1-1/n^c$, $j = 1,\cdots,n-1$. Then the polynomial root-finding problem is in $\mathcal{NC}$, in the arithmetic sense, for any $p(z) \in \mathcal{P}_{c,a}$.*

COROLLARY 3.3. *The root-finding problem for the polynomial $p(z)$ is in $\mathcal{NC}$ if, given the coefficients of $p(z)$, it is possible to compute in polylogarithmic time the coefficients of a polynomial $a(z)$ such that $|a(z_{j+1})/a(z_j)| \le 1 - 1/n^c$, $j = 1,\cdots,n-1$, where $c > 0$ is a constant and z_i, $i = 1,\cdots,n$, are the zeros of $p(z)$.*

We observe that, if there exists a general strategy to choose for the polynomial $a(z)$ such that a strong break-point exists at j, $n/4 \le j \le 3n/4$, and $a(z)$ is polylogarithmic computable, then a recursive application of Algorithm 3.1 would solve the polynomial

root-finding problem in polylogarithmic arithmetic time. In the next section we will make some considerations along these lines.

It remains to prove that Algorithm 3.1 can be performed with a floating-point arithmetic having a polynomial number of digits. This is necessary to extend the above results from the arithmetic cost to the Boolean cost. For this purpose we try to keep our computations in the set of the integers, and we evaluate an upper bound to the moduli of the integer numbers involved in the computation.

First of all, suppose that $a(z)$ is a polynomial having integer coefficients with moduli at most $2^{m'}$. It is possible to prove that $a^{(\nu)}$ is a polynomial with integer coefficients bounded in modulus by $2^{\rho(n,m,m',\mu)}$, where $\rho(n,m,m',\mu)$ is a polynomial in n,m,m', $\mu = 2^{\nu}$. Therefore, stage 1 of Algorithm 3.1 can be performed in integer arithmetic, with no error, and by using $\rho(n,m,m',\mu)$ digits.

At stage 2 the quantities $h_{i,j}$ are determinants of integer matrices; therefore, they are integers whose moduli can be bounded by means of a Hadamard-like inequality, such as

$$|h_{i,j}| \leq (||R_{i,j}||_{\infty})^{i+j} \leq (i+j)2^{(i+j)\rho(n,m,m',\mu)} \leq 2^{2n\rho(n,m,m',\mu)}.$$

Therefore, stage 2 can be performed with no rounding errors by using $f = 2n\rho(n,m,m',\mu)$ binary digits.

At stage 3 the rational numbers α_i and β_i can be computed as pairs of integers having no more than $3f+1$ binary digits by means of the relations (3.1).

Concerning the coefficients of the polynomial $t_{n-i+1}(z)$ at stage 4, we observe that the matrix

$$\tilde{T}_{n-i+1} = \operatorname{diag}(h^2_{j-2,j-1}h_{j-1,j-1}h_{j-1,j})T_{n-i+1}$$

has integer entries having at most $4f+1$ binary digits. Here $\operatorname{diag}(\gamma_i)$ denotes a diagonal matrix having diagonal entries $\gamma_1, \gamma_2, \cdots$. Therefore the polynomial

$$\tilde{t}_{n-i+1}(z) = \det \operatorname{diag}(h^2_{j-2,j-1}h_{j-1,j-1}h_{j-1,j})(T_{n-i+1} - zI)$$

$$= \prod_{j=1}^{n-i+1} (h^2_{j-2,j-1}h_{j-1,j-1}h_{j-1,j})t_{n-i+1}(z)$$

has integer coefficients having at most $(n-i+1)(4f+1)$ digits. The coefficients of $t_{n-i+1}(z)$ are rational numbers that can be computed as pairs of integer numbers having at most $(n-i+1)(4f+1)$ digits. Their floating-point representations can be computed by performing a division between floating-point numbers having $(n-i+1)(4f+1)$ digits each.

Therefore, we may conclude that, if there is at least one strong break-point, it is possible to approximate a factor of $p(z)$ within the given precision, with no rounding errors and with a polynomial number of binary digits in the arithmetic.

We recall (see [2]) that in order to compute the zeros of $p(z)$ within d binary digits of absolute precision, it is sufficient to approximate the coefficients of the monic factors of the polynomial $p(z)$ with absolute error less than $2^{-\gamma}$, $\gamma = n(m+d+2\log n+4)$. Therefore Algorithm 3.1 can be applied recursively in polylogarithmic time by using an arithmetic with a polynomial number of digits.

4. Break-point strategies. We show a strategy for choosing the polynomial $a(z)$ in such a way that a strong break-point is obtained. We consider, for simplicity, the case in which the coefficients of the monic polynomials $p(z)$ and $a(z)$ belong to the complex field.

We prove that a strong break-point exists for at least one choice of $a(z)$ in the set $\{z-g, z-g-|p(g)|^{1/n}, z-g-\underline{i}|p(g)|^{1/n}\}$, where $g = -p_{n-1}/n$ is the center of gravity of the zeros of $p(z)$ and $\underline{i}$ is the imaginary unit.

If no break-point exists for $a(z) = z - g$, then all the zeros of $p(z)$ are located inside an annulus centered in g with radii $\rho_1 < \rho_2$ such that $\rho_1/\rho_2 > (1 - 1/n^c)^{n-1}$. Choose $c = 2$ so that we have $\rho_1/\rho_2 > 0.83$ if $n > 4$, that is, the annulus has relative width less than 0.17.

In this case, the point $r = g + |p(g)|^{1/n}$ lies inside this annulus, and we may again apply Algorithm 3.1 with $a(z) = z - r$.

Suppose that also with this choice we have no strong break-point. Then the zeros of $p(z)$ are located in the intersection of two annuli having relative width less than 0.17, centered in g and in r, respectively, as in Fig. 1.

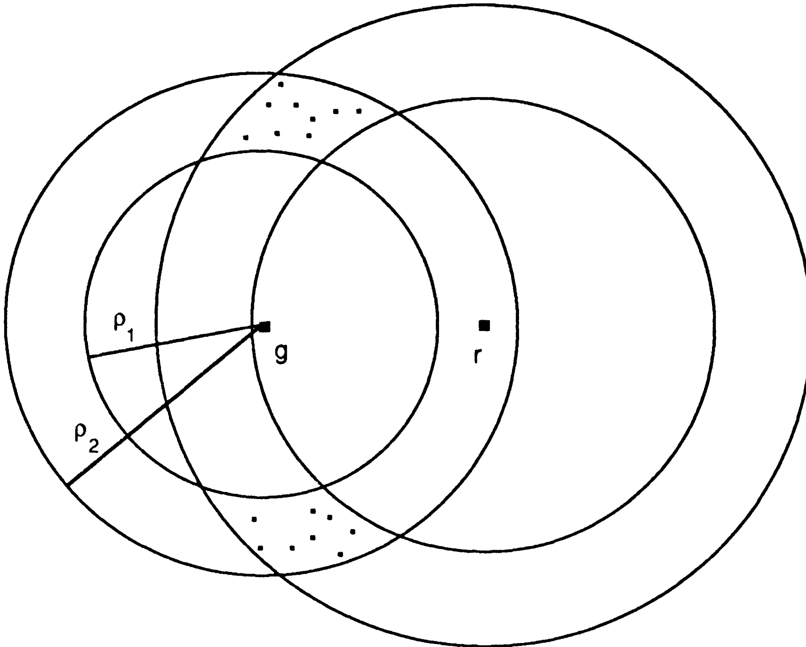


FIG. 1. *Zeros in the intersection of two annuli.*

Now, recall that g is the center of gravity of the zeros. Therefore the greatest radius of the annulus centered in r cannot be less than $|r - (g + \underline{i}\rho_1)|$, and the smallest radius cannot be greater than $|r - (g + \underline{i}\rho_2)|$. This fact can be used to prove that choosing $a(z) = z-(g+\underline{i}(r-g))$ yields a strong break-point, which allows us to split the polynomial $p(z)$ into two factors having roughly the same degree.

In the case where the polynomial has real coefficients it is possible to modify this strategy in the following way. Once we detect that the zeros are located in the intersection of two annuli, instead of choosing $a(z) = z - (g + \underline{i}(r - g))$, which would yield complex factors, we choose $a(z) = z^2 - 2gz + g^2 + (r - g)^2$, which has zeros $w = g + \underline{i}(r - g)$ and $\bar{w} = g - \underline{i}(r - g)$. If even with this choice we have no break-

point, it means that the zeros are located on the annuli determined by the two Cassini ovals $\{z \in \mathbf{C} : |a(z)| = \rho'\}$ and $\{z \in \mathbf{C} : |a(z)| = 0.83\rho'\}$ (see Fig. 2); that is, $0.83\rho' \le \sigma_j \le \rho'$, where $\sigma_j = |(z_j - w)(z_j - \bar{w})|$. Now it is possible to compute, at a low cost, a point inside these annuli. In fact, observe that $0.83\rho' \le \sigma \le \rho'$, where $\sigma = |\prod_{i=1}^{n} \sigma_i|^{1/n} = |p(w)|^{2/n}$, so the following points lie in the annuli determined by Cassini's ovals (see Fig. 3):

$$v_1 = w + ((\sigma^2 + 4(r-g)^4)^{1/2} - 2(r-g)^2)^{1/2},\ v_2 = w + \underline{i}(g - r + (\sigma + (r-g)^2))^{1/2}.$$

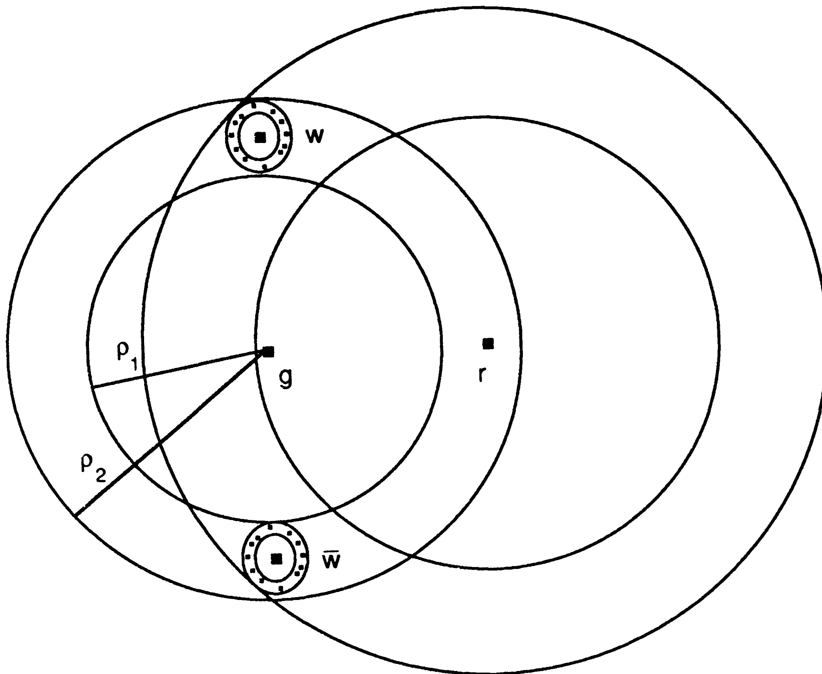


FIG. 2. *Zeros in the Cassini ovals.*

Now, since g is also the center of gravity of the real parts of the zeros of $p(z)$, and since there exist j and k such that $|g - z_j| > |w - g|$, and $|g - z_k| < |w - g|$ (recall that $w = (\prod_{i=1}^{n} |z_i - g|)^{1/n}$), at least one of the following polynomials gives a strong break-point:

$$a(z) = (z - v_1)(z - \bar{v}_1), \quad a(z) = (z - v_2)(z - \bar{v}_2).$$

This strategy does not necessarily guarantee that the break-point yields two factors of degree roughly $n/2$. For instance, choosing $a(z) = z - g$ might lead to just one break-point corresponding to two factors of degrees 1 and $n-1$, respectively. Indeed we could apply the same algorithm to the factor of degree $n-1$ and go forth. In the worst case situation the algorithm stops after $n - 1$ steps and therefore the overall cost of the algorithm is not polylogarithmic. In this case we have $|z_i - g_k| < \epsilon_k |g_k - g_{k+1}|$, $i = k+1, \cdots, n$, $k = 0, 1, \cdots, n-1$, where g_k is the gravity center of the zeros $z_{k+1}, \cdots, z_n$ and $\epsilon_k < \epsilon$ are positive numbers small enough to yield just one strong break-point at each application of the algorithm. In fact, observe that for $i, j > k$ the zeros z_i and z_j belong to the kth

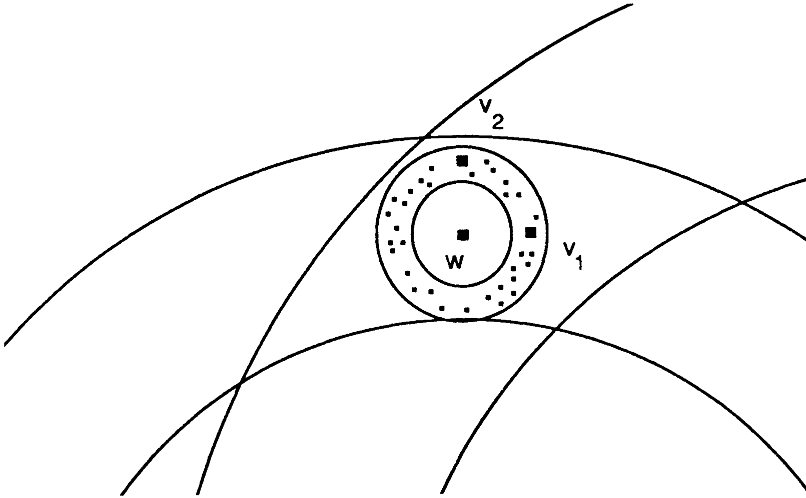


FIG. 3. *Zeros clustered in the Cassini annulus.*

cluster, being

$$(4.1) \qquad 1 - 2\epsilon + O(\epsilon^2) < |z_j - g_{n-k+1}|/|z_i - g_{n-k+1}| < 1 + 2\epsilon + O(\epsilon^2).$$

A special situation occurs, for instance, in the case for which the zeros are almost in geometric progression, i.e., $z_1 = \eta,\ z_2 = \eta + \epsilon\gamma_1,\ z_3 = \eta + \epsilon\gamma_1 + \epsilon^2\gamma_2,\ z_4 = \eta + \epsilon\gamma_1 + \epsilon^2\gamma_2 + \epsilon^3\gamma_3, \cdots$, where ϵ is a small positive number and $\gamma_i,\ i = 1, \cdots, n-1$ are complex numbers having almost the same moduli.

This situation, considered hard in [22], can be easily handled by Algorithm 3.1, by choosing $a(z) = p'(z)$. In fact, in this case we have

$$\left|\frac{\lambda_j}{\lambda_i}\right| = \prod_{r=1,\ r\neq i,\ r\neq j}^{n} \left|\frac{z_r - z_j}{z_r - z_i}\right|,$$

so that we have $n-2$ strong break-points, and one application of Algorithm 3.1 yields all the zeros. This can be proved by splitting the above product into three parts, as follows (for simplicity, assume $i < j$):

$$\left|\frac{\lambda_j}{\lambda_i}\right| = \prod_{r=1}^{i-1} \left|\frac{z_r - z_j}{z_r - z_i}\right| \prod_{r=i+1}^{j-1} \left|\frac{z_r - z_j}{z_r - z_i}\right| \prod_{r=j+1}^{n} \left|\frac{z_r - z_j}{z_r - z_i}\right|.$$

The first product is $1 + O(\epsilon)$, the second product is $\epsilon^{(j-i)(j-i-1)/2}(1 + O(\epsilon))$, and the third product is $\epsilon^{(j-i)(n-j)}(1 + O(\epsilon))$. Hence, we obtain

$$\left|\frac{\lambda_j}{\lambda_{j-1}}\right| = \epsilon^{n-j}(1 + O(\epsilon)), \qquad j = 2, \cdots, n.$$

Also, in the case of zeros clustered as in (4.1) choosing, $a(z) = p'(z)$ yields $n-2$ strong break-points. However, this choice may fail if, at each recursive application of the algorithm with the implicit shift in the gravity center, only two factors with clustered zeros, of degree $n-2$ and 2, respectively, are computed.

This situation can be handled by choosing $a(z) = p''(z)$. In general, the case for which at each recursive application of Algorithm 3.1 two factors with clustered zeros, of degree $n-h$ and h, respectively, are computed, can be treated by choosing $a(z) = p^{(h)}(z)$. This can be proved by using the relation

$$\left|\frac{p^{(h)}(z_j)}{p^{(h)}(z_i)}\right| = \left|\frac{\sum\limits_{1\le i_1<\cdots<i_{h-1}\le n,\ i_r\ne j} p_{i_1,\cdots,i_{h-1},j}(z_j)}{\sum\limits_{1\le i_1<\cdots<i_{h-1}\le n,\ i_r\ne i} p_{i_1,\cdots,i_{h-1},i}(z_i)}\right|. \tag{4.2}$$

Here we consider, for simplicity, the case $h=2$. Suppose that at the kth recursive application of Algorithm 3.1 with the implicit shift in the gravity center, we get two clusters $F_k = \{z_{2k-1}, z_{2k}\}$ and $C_k = \{z_{2k+1}, \cdots, z_n\}$ such that

$$\begin{aligned} |z_{2k-1} - z_{2k}| &< \epsilon |f_k - g_{k-1}|, \\ |z_i - z_j| &< \epsilon |g_k - g_{k-1}|, \qquad i,j > 2k, \end{aligned}$$

where f_k and g_k are the centers of gravity of clusters F_k and C_k, respectively. Now we prove that z_{2k} is well separated by z_j, $j > k$, if $a(z) = p''(z)$.

Setting $i = 2k$, from (4.2) we have

$$\left|\frac{p''(z_j)}{p''(z_i)}\right| = \left|\prod_{s=1,\ s\ne i,j}^{n} \frac{z_s - z_j}{z_s - z_i}\right| \cdot \left|\frac{\sum\limits_{s=1,\ s\ne j}^{n} \frac{1}{(z_s - z_j)}}{\sum\limits_{s=1,\ s\ne i}^{n} \frac{1}{(z_s - z_i)}}\right|.$$

Observe that, for our assumptions it follows that

$$\prod_{s=1}^{i-2} \left|\frac{z_s - z_j}{z_s - z_i}\right| = 1 + O(\epsilon).$$

Therefore, we must consider the remaining factor, which we rewrite in the following form, where the integer q is such that $|z_q - z_j| = \min_{s\ne j} |z_s - z_j|$:

$$\left(\prod_{s=i+1\ s\ne j,q}^{n} \left|\frac{z_s - z_j}{z_s - z_i}\right|\right) \left|\frac{z_{i-1} - z_j}{z_{i-1} - z_i}\right| \cdot \left|\frac{z_q - z_j}{z_q - z_i}\right| \cdot \left|\frac{\sum\limits_{s=1,\ s\ne j}^{n} \frac{1}{(z_s - z_j)}}{\sum\limits_{s=1,\ s\ne i}^{n} \frac{1}{(z_s - z_i)}}\right|. \tag{4.3}$$

Now, it is easy to prove that the first factor in (4.3) is less than

$$\left(\frac{\epsilon}{(n/2 - k)(1-\epsilon)}\right)^{n-2k-1}.$$

Concerning the remaining part, since $|(z_q - z_j)/(z_s - z_j)| \leq 1$, we obtain

$$\left|\frac{z_{i-1} - z_j}{z_q - z_i}\right| \cdot \left|\frac{\sum_{s=1,\ s\neq j}^{n} \frac{z_q - z_j}{z_s - z_j}}{1 + \sum_{s=1,\ s\neq i,i-1}^{n} \frac{z_{i-1} - z_i}{z_s - z_i}}\right| \leq \frac{1+\epsilon}{1-\epsilon} \cdot \frac{n-1}{1-O(\epsilon)},$$

so that we may conclude with the bound

$$\left|\frac{p''(z_j)}{p''(z_i)}\right| \leq (n-1)\left(\frac{2\epsilon}{n-2k}\right)^{n-2k-1} + O(\epsilon).$$

In the case where $p(z)$ has real zeros it is sufficient to find a real number w such that $n/4 \leq \#\{z_i : z_i < w\} < 3n/4$ in order to obtain factors with degree less than $3n/4$. In fact, choose $a(z) = z - w$ and apply Algorithm 3.1. If a factor $q(z)$ of degree $m > 3n/4$ is obtained, we have that at least m zeros are inside a thin annulus centered in w. Observe that $r = w + |q(w)|^{1/m}$ is inside the annulus. Choosing $a(z) = z - r$ allows us to determine only factors with degree not greater than $3n/4$. The determination of w can be performed in polylogarithmic time by applying the Euclidean algorithm to $p(z)$ and $p'(z)$ [2].

In the general case of polynomials with complex zeros it is an open problem to determine a strategy of choosing $a(z)$ such that the recursive application of the algorithm can be carried out in a polylogarithmic number of steps and $a(z)$ is polylogarithmically computable. Indeed, such a strategy can be easily devised if a separator w, dividing the real parts of the zeros into two sets having almost the same cardinality, is computable in polylogarithmic time.

Acknowledgment. The first author wishes to thank Victor Pan for helpful discussions and precious suggestions.

REFERENCES

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1976.

[2] M. Ben-Or, E. Feig, D. Kozen, and P. Tiwari, *A fast parallel algorithm for determining all roots of a polynomial with real roots*, SIAM J. Comput., 17 (1988), pp. 1081–1092.

[3] D. Bini and L. Gemignani, *On the Euclidean scheme for polynomials having interlaced real zeros*, Proc. of 2nd Ann. ACM SPAA Symp. Crete, Greece, 1990, pp. 254–258.

[4] D. Bini and V. Pan, *Improved parallel polynomial division*, SIAM J. Comput., to appear.

[5] ———, *Polynomial division and its computational complexity*, J. Complexity, 2 (1986), pp. 179–203.

[6] A. Borodin, J. von zur Gathen, and J. Hopcroft, *Fast parallel matrix and GCD computation*, Inform. and Control, 52 (1982), pp. 241–256.

[7] R. Brent, F. Gustavson, and D. Yun *Fast solution of Toeplitz systems of equations and computations of Padé approximants*, J. Algorithms, 1 (1980), pp. 259–295.

[8] L. Gemignani, *Metodi numerici per il calcolo simultaneo degli zeri di un polinomio*, Tesi di Laurea in Matematica, Department of Mathematics, University of Pisa, Italy, 1987.

[9] J. von zur Gathen, *Parallel algorithms for algebraic problems*, SIAM J. Comput., 13 (1984), pp. 802–824.

[10] A.S. Householder, *The Numerical Treatment of a Single Nonlinear Equation*, McGraw-Hill, New York, 1970.

[11] ———, *Generalization of an algorithm of Sebastiao e Silva*, Numer. Math., 16 (1971), pp. 375–382.

[12] P. Henrici, *Applied and Computational Complex Analysis*, Vol 1., John Wiley, New York, 1974.

[13] D.E. Knuth, *The Art of Computer Programming: Seminumerical Algorithms*, Vol. 2, Addison-Wesley, Reading, MA, 1981.

[14] K. MAHLER, *An inequality for the discriminant of a polynomial*, Mich. Math. J., 11 (1964), pp. 257–262.

[15] C.A. NEFF, *Specified precision polynomial root isolation is in* NC, Proc. 31st Annual IEEE Symp., 1990, pp. 138–145.

[16] YU. OFMAN, *On the algorithmic complexity of discrete functions*, Soviet Physics-Dokl., 7 (1963), pp. 589–591.

[17] V. PAN, *Sequential and parallel complexity of approximate evaluation of polynomial zeros*, Comput. Math. Appl., 14 (1987), pp. 591–622.

[18] ———, *Algebraic complexity of computing polynomial zeros*, Comput. Math. Appl., 14 (1987), pp. 285–304.

[19] V. PAN AND J. REIF, *Displacement structure and processor efficiency of fast parallel Toeplitz computations*, Proc. 28th Annual IEEE Symposium on the Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1987, pp. 173–184.

[20] S. SMALE, *Algorithms for solving equations*, Proc. Intern. Congress Math., Berkeley, CA, 1987.

[21] ———, *On the efficiency of the algorithms of analysis*, Bull. Amer. Math. Soc., 13 (1985), pp. 87–121.

[22] A. SCHÖNHAGE, *The fundamental theorem of algebra in terms of computational complexity*, Department of Mathematics, University of Tubingen, Federal Republic of Germany, 1982.

SIAM J. COMPUT.
Vol. 21, No. 5, pp. 801-823, October 1992
© 1992 Society for Industrial and Applied Mathematics
001

INCREASING THE SIZE OF A NETWORK BY A CONSTANT FACTOR CAN INCREASE PERFORMANCE BY MORE THAN A CONSTANT FACTOR*

RICHARD KOCH†

Abstract. The performance of unbuffered routing algorithms for parallel computer architectures is analyzed. Unbuffered algorithms are an alternative to the use of queues. When the capacity of a switch or communications link is exceeded, the extra messages are discarded, and another attempt to transmit the message will be made at a later time. The analysis presented here is relevant for routing on the BBN Butterfly and Agarwal and Knight's Alewife architecture, both of which have interconnection networks based on the butterfly graph.

Suppose that each of the N inputs of the butterfly independently decides to send a message with probability p, and that the message is sent to a random output, with each output having an equal probability of being chosen. If more than q messages attempt to traverse an edge, the extra messages over q are discarded. q is called the dilation of the network. The bandwidth is the number of messages that reach their destinations. It is shown that if $p = \Omega((\log N)^{-1/q})$, the expected bandwidth is $\Theta(N(\log N)^{-1/q})$, and if $p = o((\log N)^{-1/q})$, the expected bandwidth is $N(p + o(p))$. This result also holds for dilated networks based on the d-ary butterfly and for graphs constructed by taking r copies of butterflies and identifying corresponding inputs and also identifying corresponding outputs. An expression is also derived for the asymptotic constants and it is shown that the probability distribution is tightly concentrated about its mean. Interesting techniques are developed for finding asymptotics of nonlinear systems of recurrences.

The result may also have implications for design trade-offs since, for sufficiently large networks, having a fixed amount of hardware increasing the value of q will increase bandwidth more than increasing the values of d or r.

Key words. parallel computation, interconnection network, parallel computer architecture, performance analysis, difference equations

AMS(MOS) subject classification. 68M10

1. Introduction.

When designing computer architectures it is useful to have quantitative measures of system performance. Since the performance of actual systems cannot be measured until the system has been built, it is necessary to resort to analytical and simulation studies when making design decisions. Unbuffered routing strategies for multistage interconnection networks for multiprocessor systems have been proposed and analyzed in several previous papers [8], [5], [6], [3]. An interconnection network that uses such a strategy is being built at MIT by Knight and Agarwal [4], and the BBN Butterfly uses a related routing strategy [2]. With unbuffered routing, congestion is eliminated by discarding messages rather than holding excess messages in queues; a mechanism is provided for notifying a processor if its message is discarded, and, if necessary, the message is retransmitted at a later time.

We will analyze unbuffered routing on the butterfly network. If N, σ are integers such that $N = 2^\sigma$, then the butterfly network of size N is defined in the following way. There are three kinds of nodes: input nodes, which have degree one, where processors are located; output nodes, which have degree one, where processors are located; and

* Received by the editors December 26, 1989; accepted for publication (in revised form) July 25, 1991. This research was supported by a National Science Foundation graduate fellowship, an Army Science and Technology fellowship, Defense Advanced Research Projects Agency contract N00014-87-K-825, Office of Naval Research contract N00014-86-K-0593, Air Force contract OSR-86-0076, and Army contract DAAL-03-86-K-0171.

† Department of Mathematics and Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. Present address, AT&T Bell Laboratories, 101 Crawfords Corner Road, Holmdel, New Jersey 07733 (rrk@hoqaa.att.com).

switching nodes, which have degree four and where switches are located. Input nodes are labelled with integers k such that $0 \leqq k \leqq N-1$; output nodes are labelled similarly. Switching nodes are labelled with ordered pairs of integers (l, m), where $0 \leqq l \leqq (N-1)/2$ and $1 \leqq m \leqq \sigma$. Let $b_{\sigma-1} \cdots b_1$ be the binary representation of l. Then for a switching node (l, m) with $m < \sigma$ and for $i = 0, 1$, let l_i be the integer with binary representation $b_{\sigma-1} \cdots b_{m+1} i b_{m-1} \cdots b_1$. Then there is an edge (physically realized with a wire) between (l, m) and $(l_i, m+1)$. In addition, for switching nodes with $m = 1$ there are edges connecting (l, m) to input nodes whose labels have binary representation $b_{\sigma-1} \cdots b_1 i$ for $i = 0$, 1 and for switching nodes with $m = \sigma$ there are edges connecting (l, m) to output nodes whose labels have binary representation $b_{\sigma-1} \cdots b_1 i$ for $i = 0$, 1, (see Fig. 1).

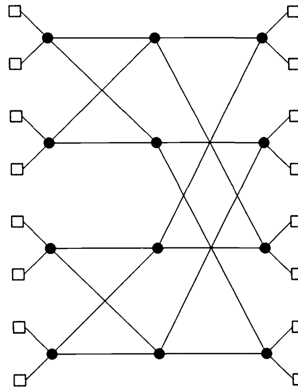


FIG. 1. *Butterfly network of size* 8.

The butterfly network is known by other names such as the FFT network, the multistage cube [12], the indirect binary n-cube [9], and the generalized cube [13] and is isomorphic to other networks such as the omega network [7] and the baseline network [14]. It is an example of a delta network [8].

Before we analyze the performance of unbuffered routing, we first make a few definitions. All input nodes are defined to have level zero; all output nodes are defined to have level $\sigma+1$, and a switching node (l, m) is defined to have level m. We denote by V_m the set of all nodes with level m. We denote by E_m the set of all edges connecting nodes with level m to nodes with level $m+1$. For $e \in E_m$, where $m > 1$, we define e_0 and e_1 to be the two edges of E_{m-1} incident to e.

Now suppose that each processor at an input node independently decides to make a request to a memory at some output node with probability p, with each output node having an equal probability of being chosen. At each time $m = 0, \cdots, \sigma$ packets attempt to advance from the nodes where they are currently located in V_m to the next nodes on their paths to their destinations in V_{m+1}. If less than or equal to q packets attempt to traverse an edge in E_m, all of the packets are forwarded to the next node in V_{m+1}. If more than q packets attempt to traverse an edge in E_m, q of the packets are forwarded to the next node in V_{m+1}, and the other packets are discarded. The packets that are discarded are chosen in such a way that the choice is independent of the packet destinations. We will refer to q as the dilation of the butterfly network. For an edge e, we let X_e be the number of packets that traverse e. Let

$$S_m = \sum_{e \in E_m} X_e$$

Then we define the bandwidth to be S_σ; the bandwidth is just the number of packets that successfully reach their destinations.

The question of finding the bandwidth when $q=1$ was first raised by Patel [8], who derived a recurrence to describe the bandwidth. Kruskal and Snir [5] found the asymptotics for this recurrence. They also raised the problem of finding the bandwidth butterfly networks with $q \geqq 2$ and derived a system of recurrences to describe the bandwidth. Kumar and Jump [6] conjectured that dilated networks have bandwidth $\Theta(N(\log N)^{-1/q})$ based on a plausible but unproved assumption about a particular probability distribution; although the assumption resulted in a correct estimate up to constant factors, their assumption resulted in an incorrect evaluation of the asymptotic constant. Knight [3] has also considered the problem of computing generating functions to describe the bandwidth of unbuffered routing.

2. The principal theorem. The following theorem holds.

THEOREM 1. *When $q=1$, expected bandwidth is*

$$N(\tfrac{1}{4}\log N + p^{-1} + O(\log\log N))^{-1}$$

and for constant $q \geqq 2$, when $p = o((\log N)^{-1/q})$, the expected bandwidth is

$$Np(1+O(p^q \log N + p)),$$

and if $p = \Omega((\log N)^{-1/q})$, then the expected bandwidth is

$$N(p^{-q} + \chi_q \log N)^{-1/q} + O(N(\log N)^{-2/q}),$$

where

$$\chi_q = \frac{q(1-2^{-q})}{(q+1)!}.$$

For example, when $q=2$ and p is a constant, the expected bandwidth is asymptotic to

$$\frac{2N}{\sqrt{\log N}}.$$

Increasing q from 1 to 2 increases the expected bandwidth by $\Theta(\sqrt{\log N})$. Thus, increasing the amount of hardware by a constant factor can increase bandwidth by more than a constant factor! If we look at the bandwidth as p varies from zero to one, we observe a threshold. When p is $\omega((\log N)^{-1})$, the bandwidth is asymptotic to $4N/\log N$, and when p is $o((\log N)^{-1})$, the bandwidth is asymptotic to pN, in which case the expected fraction of packets that successfully reach their destination approaches 1 as $N \to \infty$. However, when p is $\omega((\log N)^{-1})$, the expected bandwidth is virtually independent of p; attempting to send more packets results in few additional packets reaching their destination. If γ is a constant and p is $\gamma(\log N)^{-1}$, a constant fraction of the packets reach their destinations, where the constant depends on γ.

3. Asymptotics for a nonlinear recurrence. To prove Theorem 1, the following theorem of de Bruijn [1] will be essential. Since we need an estimate of an error term that was not given in [1], we prove the result here.

THEOREM 2. *Suppose q is a fixed positive integer and*

$$\lim_{m\to\infty} s_m = 0, \qquad 0 \leqq s_0 \leqq 1, \qquad c>0,$$

$$s_{m+1} = s_m - cs_m^{q+1} + O(s_m^{q+2}).$$

Then, if $q=1$,

$$s_m=(cm+s_0^{-1}+O(\log m))^{-1},$$

and if $q\geqq 2$,

$$s_m=(cqm+s_0^{-q}+O(m^{1-1/q}))^{-1/q}.$$

Proof. Let $t_m=s_m^q$. Then

$$\begin{aligned} t_{m+1}^{1/q}&=t_m^{1/q}-ct_m^{1+(1/q)}+O(t_m^{1+(2/q)}),\\ t_{m+1}&=t_m(1-ct_m+O(t_m^{1+(1/q)}))^q\\ &=t_m(1-cqt_m+O(t_m^{1+(1/q)})). \end{aligned}$$

Let $u_m=t_m^{-1}$. Then

$$\begin{aligned} u_{m+1}&=u_m(1-cqu_m^{-1}+O(u_m^{-1-(1/q)}))^{-1}\\ &=u_m+cq+O(u_m^{-(1/q)}), \end{aligned}$$

and thus

$$u_m=cqm+u_0+\sum_{i=1}^m O(u_i^{-1/q}).$$

Since

$$\lim_{m\to\infty} s_m=0,\quad \lim_{m\to\infty} u_m^{-1/q}=0,\quad u_m=\Omega(m),$$

$$\sum_{i=1}^m O(u_i^{-1/q})=O\left(\sum_{i=1}^m i^{-1/q}\right)=\begin{cases} O(\log m) & q=1,\\ O(m^{1-(1/q)}) & q\geqq 2.\end{cases}$$

Thus if $q=1$,

$$\begin{aligned} u_m&=cm+u_0+O(\log m),\\ s_m&=(cm+s_0^{-1}+O(\log m))^{-1}, \end{aligned}$$

and if $q\geqq 2$,

$$\begin{aligned} u_m&=cqm+u_0+O(m^{1-(1/q)}),\\ s_m&=(cqm+s_0^{-q}+O(m^{1-(1/q)}))^{-1/q}. \end{aligned}$$

□

4. Proof of the principal theorem for $q=1$. The proof of Theorem 1 is long and technical. In order to give an understanding of the underlying ideas, we first prove the cases $q=1$ and $q=2$, and then give a sketch of the proof for the general theorem before giving the technical details.

Proof of Theorem 1 *when* $q=1$. For $m=0,\cdots,\sigma$ let $p_m=\Pr\{X_e=1\}$, where $e\in E_m$; by symmetry, the definition of p_m will not depend on the choice of e. Packets that traverse an edge $e\in E_{m+1}$ must have traversed either e_0 or e_1 at the previous step. Packets that traverse e_0, e_1 will attempt to traverse e with probability $\frac{1}{2}$. Since the set of input nodes that can send their packets through e_0 is disjoint from the set of input nodes that can send their packets through e_1, X_{e_0} and X_{e_1} are independent. If $X_{e_0}, X_{e_1}=1$, then $X_e=1$ with probability $\frac{3}{4}$; if $X_{e_0}=1, X_{e_1}=0$ or $X_{e_0}=0, X_{e_1}=1$, then $X_e=1$ with probability $\frac{1}{2}$, and if $X_{e_0}, X_{e_1}=0$ then clearly $X_e=0$. Thus

$$\begin{aligned} p_{m+1}&=\tfrac{3}{4}p_m^2+\tfrac{1}{2}p_m(1-p_m)+\tfrac{1}{2}p_m(1-p_m)\\ &=p_m-\tfrac{1}{4}p_m^2. \end{aligned}$$

We now show that $p_m \to 0$ when $m \to \infty$. It clearly follows from the recurrence for p_m that $p_{m+1} \leqq p_m$, and since we know that $p_m \geqq 0$, it follows that there exists an L such that $p_m \to L$ since any bounded monotone sequence converges [10]. Then

$$\begin{aligned} L &= \lim_{m\to\infty} p_{m+1} \\ &= \lim_{m\to\infty} \left(p_m - \tfrac{1}{4}p_m^2\right) \\ &= L - \tfrac{1}{4}L^2, \end{aligned}$$

from which it follows that $L = 0$. It follows from Theorem 2 that

$$p_m = (p^{-1} + \tfrac{1}{4}m + O(\log m))^{-1}.$$

Thus

$$\begin{aligned} E(S_\sigma) &= E\left(\sum_{e\in E_\sigma} X_e\right) \\ &= \sum_{e\in E_\sigma} E(X_e) \\ &= \sum_{e\in E_\sigma} p_\sigma \\ &= Np_\sigma. \end{aligned}$$

□

We now continue with the proof of Theorem 1 when $q \geqq 2$. X_e and S_m are defined as before. If $e, e' \in E_m$, then by symmetry

$$\Pr\{X_e = i\} = \Pr\{X_{e'} = i\}.$$

Thus we can define

$$p_{m,i} = \Pr\{X_e = i\}, \qquad e \in E_m.$$

It follows that $p_{0,1} = p$ and $p_{0,i} = 0$, $i = 2, \cdots, q$. Choose $e \in E_m$; we define z_m by

$$\begin{aligned} z_m &= E(X_e) \\ &= \sum_{i=1}^{q} ip_{m,i}. \end{aligned}$$

Since

$$\begin{aligned} E(S_m) &= E\left(\sum_{e\in E_m} X_e\right) \\ &= \sum_{e\in E_m} E(X_e) \\ &= Nz_m, \end{aligned}$$

we know that the expected bandwidth is Nz_σ.

5. Proof of the principal theorem for $q = 2$. We first prove the case $q = 2$.

Proof of Theorem 1 *when* $q = 2$. We first wish to write recurrences for $p_{m,1}$ and $p_{m,2}$. Suppose $e \in E_{m+1}$. We know that X_{e_0} and X_{e_1} are independent since the set of input nodes that may send packets through e_0 is disjoint from the set of input nodes that may send packets through e_1. If j packets traverse e_0 and k packets traverse e_1, then since each of these $j + k$ packets will traverse e with probability $\frac{1}{2}$, the probability

$\gamma_{i,j,k}$ that i packets will traverse e is

$$\binom{j+k}{i}2^{-j-k}, \quad i<q; \qquad \sum_{s=i}^{j+k}\binom{j+k}{s}2^{-j-k}, \quad i=q.$$

Thus

$$p_{m+1,i}=\sum_{0\leqq j,k\leqq 2}\gamma_{i,j,k}p_{m,j}p_{m,k}.$$

If $p_{m,0}$ is replaced by $1-p_{m,1}-p_{m,2}$ and similar terms are collected, we have the following recurrence relations:

$$p_{m+1,1}=p_{m,1}+p_{m,2}-\tfrac{1}{2}p_{m,1}^2-\tfrac{3}{4}p_{m,2}^2-\tfrac{5}{4}p_{m,1}p_{m,2}, \tag{1}$$

$$p_{m+1,2}=\tfrac{1}{2}p_{m,2}+\tfrac{1}{4}p_{m,1}^2+\tfrac{3}{16}p_{m,2}^2+\tfrac{1}{2}p_{m,1}p_{m,2}. \tag{2}$$

It follows that

$$z_{m+1}=z_m-\tfrac{3}{8}p_{m,2}^2-\tfrac{1}{4}p_{m,1}p_{m,2}. \tag{3}$$

We would like to show that z_m satisfies the recurrence

$$z_{m+1}=z_m-\Theta(z_m^3),$$

from which we could conclude that z_m is $(p^{-2}+\Theta(m))^{-1/2}$ by Theorem 2, but it is not obvious how to prove that z_m satisfies such a recurrence. We will define $z_{m,1}$ and $z_{m,2}$, in a way that preserves the asymptotic behavior of z_m, and so that we get a $z_{m,2}$ for which we can show the desired recurrence. We define $z_{m,1}$ by

$$z_{m,1}=p_{m,1}+2p_{m,2}-\tfrac{1}{2}p_{m,1}p_{m,2}.$$

We now wish to show that $z_{m,1}\sim z_m$. We need the following lemma.

LEMMA 1. $\lim_{m\to\infty} z_m=0$.

Proof of Lemma 1. Clearly $z_{m+1}\leqq z_m$ and $z_m\geqq 0$. Thus there exists an L such that $\lim_{m\to\infty} z_m=L$. Suppose $L>0$, and choose

$$\varepsilon<\min\{\tfrac{1}{24}L^2, \tfrac{1}{3456}L^4\}.$$

Then there exists M such that if $m\geqq M$, then $L\leqq z_m<L+\varepsilon$. We consider two cases.

Case 1. $p_{M,2}>\frac{1}{3}L$. It follows from (3) that

$$\begin{aligned} z_{M+1} &\leqq z_M-\tfrac{3}{8}p_{M,2}^2 \\ &< L+\varepsilon-\tfrac{1}{24}L^2 \\ &< L. \end{aligned}$$

Case 2. $p_{M,2}\leqq\frac{1}{3}L$. Then $p_{M,1}>\frac{1}{3}L$. It follows by (2) that $p_{M+1,2}>\frac{1}{36}L^2$. It follows from (3) that

$$\begin{aligned} z_{M+2} &\leqq z_{M+1}-\tfrac{3}{8}p_{M+1,2}^2 \\ &< L+\varepsilon-\tfrac{1}{3456}L^4 \\ &< L. \end{aligned}$$

Since $\lim_{m\to\infty} z_m=0$, it follows that $\lim_{m\to\infty} p_{m,1}, p_{m,2}=0$, and thus

$$p_{m,1}p_{m,2}=O(\max\{p_{m,1}, p_{m,2}\}^2),$$

and since

$$p_{m,1}+2p_{m,2}=\Theta(\max\{p_{m,1}, p_{m,2}\}),$$

it follows that $z_m \sim z_{m,1}$. We find a recurrence for $z_{m,1}$ in the following way:

$$z_{m+1,1} = z_{m+1} - \tfrac{1}{2} p_{m+1,1} p_{m+1,2}$$
$$= z_m - \tfrac{3}{8} p_{m,2}^2 - \tfrac{1}{4} p_{m,1} p_{m,2} - \tfrac{1}{2} p_{m+1,1} p_{m+1,2}.$$

Now replace z_m by $z_{m,1} + \frac{1}{2} p_{m,1} p_{m,2}$ to get the recurrence

$$z_{m+1,1} = z_{m,1} - \tfrac{3}{8} p_{m,2}^2 + \tfrac{1}{4} p_{m,1} p_{m,2} - \tfrac{1}{2} p_{m+1,1} p_{m+1,2}.$$

Replace $p_{m+1,1}$ and $p_{m+1,2}$ by the right-hand side of the recurrences (1) and (2), respectively, and collect similar terms to yield the following recurrence:

$$z_{m+1,1} = z_{m,1} - \tfrac{5}{8} p_{m,2}^2 - \tfrac{1}{8} p_{m,1}^3 - \tfrac{1}{4} p_{m,1}^2 p_{m,2} - \tfrac{1}{32} p_{m,1} p_{m,2}^2 + \tfrac{3}{32} p_{m,2}^3 + P_1(p_{m,1}, p_{m,2}),$$

where P_1 is a polynomial with no monomials of degree less than or equal to three. $z_{m,1}$ was defined so that there would be no $p_{m,1} p_{m,2}$ term in the recurrence for $z_{m,1}$. The reason for doing this will be made clear later. Now let

$$z_{m,2} = z_{m,1} - \tfrac{5}{6} p_{m,2}^2.$$

$z_{m,2} \sim z_m$ by an argument similar to that used to prove that $z_{m,1} \sim z_m$. Elementary calculations similar to those used to derive the recurrence for $z_{m,1}$ give the following recurrence for $z_{m,2}$:

$$z_{m+1,2} = z_{m,2} - \tfrac{1}{8} p_{m,1}^3 - \tfrac{11}{24} p_{m,1}^2 p_{m,2} - \tfrac{43}{96} p_{m,1} p_{m,2}^2 - \tfrac{1}{16} p_{m,2}^3 + P_2(p_{m,1}, p_{m,2}), \tag{4}$$

where P_2 is a polynomial with no monomials of degree less than or equal to three. $z_{m,2}$ was defined so that there would be no $p_{m,2}^2$ term in the recurrence for $z_{m,2}$.

Now, since $z_{m,2} = \Theta(\max\{p_{m,1}, p_{m,2}\})$, it follows that

$$z_{m+1,2} = z_{m,2} - \Theta(z_{m,2}^3).$$

Thus we can conclude from Theorem 2 that $z_{m,2}$ is $(p^{-2} + \Theta(m))^{-1/2}$, and since $z_m = z_{m,2} + O(z_{m,2}^2)$,

$$z_m = (p^{-2} + \Theta(m))^{-1/2} + O((p^{-2} + m)^{-1}),$$

and thus z_σ is $\Theta((\log N)^{-1/2})$ when $p = \Omega((\log N)^{-1/2})$ and is $p(1 + O(p^2 \log N + p))$ when $p = o((\log N)^{-1/2})$.

We now assume that $p = \Omega((\log N)^{-1/2})$. We first prove that $p_{m,1}$ is $\Theta(m^{-1/2})$ and $p_{m,2}$ is $\Theta(m^{-1})$. We need a lemma.

LEMMA 2. *If $0 < c < 1$, r a real number, then*

$$\sum_{s=1}^{m} c^{m-s} s^r = \Theta(m^r).$$

Proof. Let $K = \lceil -\log m / \log c \rceil$. Then

$$\sum_{s=1}^{m} c^{m-s} s^r = \sum_{s=0}^{m-1} c^s (m-s)^r$$
$$= \sum_{s=0}^{K-1} c^s m^r \left(1 + O\left(\frac{s}{m}\right)\right) + \sum_{s=K}^{m-1} c^s (m-s)^r$$
$$\leqq \sum_{s=0}^{\infty} c^s m^r + O\left(\sum_{s=0}^{K-1} s c^s m^{r-1}\right) + \sum_{s=K}^{m-1} c^K m^r$$
$$\leqq \frac{m^r}{1-c} + O(K^2 m^{r-1}) + O(m^r)$$
$$= O(m^r).$$

A simple induction on m using (2) shows that

$$p_{m+1,2} = \sum_{s=0}^{m} 2^{-(m-s)}(\tfrac{1}{4}p_{s,1}^2 + \tfrac{3}{16}p_{s,2}^2 + \tfrac{1}{2}p_{s,1}p_{s,2}). \tag{5}$$

We know that $p_{m,1}$ and $p_{m,2}$ are $O(m^{-1/2})$ because z_m is $\Theta(m^{-1/2})$, and thus it follows from Lemma 2 that $p_{m,2}$ is $O(m^{-1})$. Since z_m is a linear combination of $p_{m,1}$ and $p_{m,2}$, we know that $p_{m,1}$ is $\Theta(m^{-1/2})$. It then follows from (5) and Lemma 2 that $p_{m,2}$ is $\Theta(m^{-1})$.

We can now conclude that $p_{m,1} = z_m + O(z_m^2)$ and $p_{m,2} = O(z_m^2)$. It follows from (4) and Theorem 2 that

$$\begin{aligned} z_{m,2} &= (p^{-2} + \tfrac{1}{4}m + O(m^{1/2}))^{-1/2} \\ &= (p^{-2} + \tfrac{1}{4}m)^{-1/2}(1 + O(m^{-1/2}))^{-1/2} \\ &= (p^{-2} + \tfrac{1}{4}m)^{-1/2}(1 + O(m^{-1/2})). \end{aligned}$$

Since $z_m = z_{m,2} + O(m^{-1})$, the theorem now follows when $q = 2$ since the expected bandwidth equals Nz_σ. □

6. Sketch of the proof of the principal theorem for $q > 2$. To prove Theorem 1 for $q > 2$, we will generalize the procedure used to prove the theorem for $q = 2$. We wish to prove that z_m satisfies the recurrence

$$z_{m+1} = z_m - \Theta(z_m^{q+1}).$$

We can derive a nonlinear system of difference equations to describe a relationship between the $p_{m,i}$'s, as in (1) and (2). A recurrence can also be derived to describe a relationship between z_m and the $p_{m,i}$'s as in (3). It is not immediately clear how to show the desired recurrence for z_m from a recurrence that contains $p_{m,i}$'s since we do not have an expression for each $p_{m,i}$ in terms of z_m. We will successively define $z_{m,1}, \cdots, z_{m,s^{**}}$ so that $z_{m,s^{**}}$ has the same asymptotics as $z_{m,s}$, and we can easily show the desired asymptotics for $z_{m,s^{**}}$; for simplicity, in making assertions, we will also denote z_m by $z_{m,0}$. For $0 \leqq s \leqq s^{**}$ we will have a recurrence

$$z_{m+1,s} = z_{m,s} + A_s(p_{m,1}, \cdots, p_{m,q}),$$

where A_s is a polynomial.

$A_{s^{**}}$ will have no more monomials of degree less than or equal to q, and each monomial of degree $q+1$ will have a negative coefficient. Since for $i \geqq 1$, $p_{m,i} = O(z_m)$, and since

$$z_m = \sum_{i=1}^{q} i p_{m,i},$$

from which it follows that

$$z_m = \Theta\left(\max_{1 \leqq i \leqq q} \{p_{m,i}\}\right),$$

at least one of $p_{m,1}^{q+1}, \cdots, p_{m,q}^{q+1}$ will be $\Theta(z_m^{q+1})$. Furthermore, each monomial in $A_{s^{**}}$ of degree greater than $q+1$ is $O(z_m^{q+2})$. Since it will be true that

$$z_m = z_{m,s^{**}} + O(z_{m,s^{**}}^2),$$

we will be able to conclude that

$$z_{m+1,s^{**}} = z_{m,s^{**}} - \Theta(z_{m,s^{**}}^{q+1}).$$

It will then follow from Theorem 2 that $z_{m,s^{**}} = (p^{-q} + \Theta(m))^{-q}$.

The $z_{m,1}, \cdots, z_{m,s^{**}}$ will be defined in two stages. In the first stage we define $z_{m,s}$ for $1 \leqq s \leqq s^*$ so that A_{s^*} has no monomials of degree less than or equal to q; $z_{m,s+1}$ will be defined by modifying $z_{m,s}$ so as to force the coefficient of a monomial of degree less than or equal to q in A_{s+1} to zero. In the second stage we will define $z_{m,s}$ for $s^*+1 \leqq s \leqq s^{**}$ so that all monomials of degree less than or equal to q still have coefficient zero and all monomials of degree $q+1$ have negative coefficients; in this stage $z_{m,s+1}$ will be defined by modifying $z_{m,s}$ so as to force the coefficient of a monomial of degree $q+1$ in A_{s+1} to be negative.

In both stages we modify $z_{m,s}$ to define $z_{m,s+1}$ as follows. We let

$$z_{m,s+1} = z_{m,s} + \lambda_s \phi_s(p_{m,1}, \cdots, p_{m,q}),$$

where ϕ_s is a monomial and λ_s is chosen to force the coefficient of ϕ_s in A_{s+1} to zero or some negative number. How do we choose λ_s? Let's look again at how in the case $q=2$ we forced the coefficient of $p_{m,1}p_{m,2}$ to be zero. We start by letting

$$z_{m,1} = z_m + \lambda p_{m,1} p_{m,2}.$$

Then we derive the recurrence for $p_{m,1}$ as follows:

$$\begin{aligned} z_{m+1,1} &= z_{m+1} + \lambda p_{m+1,1} p_{m+1,2} \\ &= z_m - \tfrac{3}{8} p_{m,2}^2 - \tfrac{1}{4} p_{m,1} p_{m,2} + \lambda p_{m+1,1} p_{m+1,2} \\ &= z_{m,1} - \lambda p_{m,1} p_{m,2} - \tfrac{3}{8} p_{m,2}^2 - \tfrac{1}{4} p_{m,1} p_{m,2} + \lambda p_{m+1,1} p_{m+1,2}. \end{aligned}$$

Now replace $p_{m+1,1}$ and $p_{m+1,2}$ by the right-hand sides of the recurrences in (1) and (2), respectively. Then the coefficient of $p_{m,1}p_{m,2}$ in the resulting recurrence is $-\lambda - \frac{1}{4} + \mu\lambda$, where μ is the coefficient of $p_{m,1}p_{m,2}$ in the expression that results when the right-hand side of the recurrences (1) and (2) are multiplied together. The coefficient of $p_{m,1}p_{m,2}$ in the recurrence for $z_{m,1}$ can be forced equal to a zero by choosing

$$\lambda = -\frac{1}{4(1-\mu)},$$

which is possible as long as $\mu \neq 1$, which is true since μ is the product of the coefficient of $p_{m,1}$ on the right-hand side of the recurrence in (1), which equals 1, and the coefficient of $p_{m,2}$ in the right-hand side of the recurrence in (2), which equals $\frac{1}{2}$.

When we attempt to define the $z_{m,s}$'s so as to have the desired properties, we encounter several problems. After we force the coefficient of a monomial ϕ in A_s to be zero, how do we know that when we subsequently force the coefficient of ψ in A_t to be zero, where $s < t$, that the coefficient of ϕ in A_t does not become nonzero again? To answer this question, we need to look at the properties of a particular polynomial.

Suppose that ϕ is a monomial. Henceforth we will always assume that the polynomials are polynomials in the indeterminates $w_1, \cdots, w_q$. Let $\mathscr{P}(\phi)$ be the polynomial that results when each w_i in ϕ is replaced by the right-hand side of a recurrence that expresses each $p_{m+1,i}$ as a linear combination of $p_{m,j}$'s and $p_{m,j}p_{m,k}$'s such that $1 \leqq j, k \leqq q$, and then each $p_{m,i}$ in the resulting expression is replaced by an indeterminate w_i. Now suppose ψ is the monomial whose coefficient in the recurrence was forced equal to zero or a negative number when $z_{m,s}$ is modified to define $z_{m,s+1}$. By looking closely at the derivation of the recurrence for $z_{m,1}$ in the case $q=2$ it should

seem reasonable that the coefficient of a monomial ϕ in the recurrence for $z_{m,s}$ will differ from the coefficient of ϕ in the recurrence for $z_{m,s+1}$ only if ϕ has a nonzero coefficient in $\mathscr{P}(\psi)$.

We define the order of a monomial ϕ to be sum of all the i's of the w_i's taken according to multiplicity; in other words, the degree of the monomial that results by replacing w_i by x^i, where x is an indeterminate. We will prove that ϕ has a nonzero coefficient in $\mathscr{P}(\psi)$ only if both of the following conditions are true:

- degree of $\phi \geqq$ degree of ψ,
- order of $\phi \geqq$ order of ψ.

Furthermore, the only time that ϕ will have a nonzero coefficient in $\mathscr{P}(\psi)$ when the degree of ϕ equals the degree of ψ and the order of ϕ equals the order of ψ is when $\phi = \psi$. Then, if when defining the $z_{m,s}$'s, we first force the coefficients to zero of those monomials of degree two, starting with those with the smallest order and working up to those with the largest order, then we do the same for monomials with degree three, etc. The coefficient of a monomial ϕ will remain zero (or negative for those monomials of degree $q+1$) after it has been forced to zero because when we subsequently force the coefficient of ψ to zero, the degree of ϕ will be less than or equal to the degree of ψ or the order of ϕ will be less than or equal to the degree of ψ. Thus ϕ has a zero coefficient in $\mathscr{P}(\psi)$, and the coefficient of ϕ will be identical to what it was before the coefficient of ψ was forced to zero.

Another problem appears to occur when we attempt to force the coefficients of monomials to zero. When we found the constants necessary to force the coefficient of $p_{m,1}p_{m,2}$ to zero for the case $q=2$, it was necessary to divide by $1-\mu$. Had μ been equal to one, we would not have been able to find the desired constant. In order to force the coefficient of a monomial ϕ to zero, we need to divide by $1-\nu$, where ν is the coefficient of ϕ in $\mathscr{P}(\phi)$. $1-\nu$ will be equal to zero only if ϕ is not of the form w_1^k, where k is a positive integer.

What do we do if we need to force to zero the coefficient of a ϕ such that $\nu=1$? It turns out that this is never necessary since monomials of order less than or equal to q always have zero coefficients, and any monomial of the form w_1^k, where $k \leqq q$, has order less than or equal to q. We prove that monomials of order less than or equal to q always have zero coefficients by first proving the assertion for the coefficients of monomials in z_m. We then prove it for all s using induction on s. To prove the induction, we make use of the property of $\mathscr{P}$ that ϕ has a nonzero coefficient in $\mathscr{P}(\psi)$ only if the order of ϕ is greater than or equal to the order of ψ; thus when we force to zero the coefficient of a monomial with order greater than q, the coefficients of other monomials will change only if their order is also greater than q.

It is also necessary for the monomial w_1^{q+1} to have a negative coefficient in the recurrence for $z_{m,s^{**}}$, but again for this monomial we will not be able to force it negative using our usual technique since we cannot divide by zero. Fortunately though, it turns out that after the first stage w_1^{q+1} will already have a negative coefficient; we prove this by proving by induction on s that there always exists a monomial of order $q+1$ that has a negative coefficient. Since w_1^{q+1} is the only monomial of degree greater than or equal to $q+1$ with order $q+1$, we can conclude that w_1^{q+1} has a negative coefficient after the coefficients of all monomials of degree less than or equal to q have been forced to zero.

To evaluate asymptotic constants when $p=\Omega(N(\log N)^{-1/q})$, we first prove for $i \geqq 1$ that $p_{m,i}=\Theta(m^{-1/q})$. We then prove that $p_{m,i}=(1/i!)z_m^i+O(z_m^{i+1})$. The constants are then evaluated by replacing the $p_{m,i}$'s in the recurrence for z_m with the previous formula in order to yield a recurrence whose asymptotics follow from Theorem 2.

7. Derivation of the initial recurrences. We now provide the details of the proof of Theorem 1 for constant $q \geqq 2$. We first find a system of nonlinear recurrences in the variables $p_{m,i}$ for $1 \leqq i \leqq q$ and prove certain properties about them.

THEOREM 3. *There exist constants $\alpha_{i,j}$ and $\beta_{i,j,k}$ such that*

(1) *For $i = 1, \cdots, q$, $m = 0, \cdots, \sigma - 1$,*

$$p_{m+1,i} = \sum_{j=1}^{q} \alpha_{i,j} p_{m,j} + \sum_{\substack{1 \leqq j,k \leqq q \\ j \leqq k}} \beta_{i,j,k} p_{m,j} p_{m,k}.$$

(2) $\alpha_{i,i} = 2^{1-i}$.

(3) *If $i < j$ then $\alpha_{i,j} > 0$.*

(4) *If $i > j$ then $\alpha_{i,j} = 0$.*

(5) *If $i = j + k$ then $\beta_{i,j,k} = 2^{-i}$ if $j = k$ and $\beta_{i,j,k} = 2^{1-i}$ if $j < k$.*

(6) *If $i > j + k$ then $\beta_{i,j,k} = 0$.*

Proof. Suppose $e \in E_{m+1}$. As before, X_{e_0} and X_{e_1} are independent, and we define $\gamma_{i,j,k}$ to be the probability that i packets traverse e given that j packets traverse e_0 and k packets traverse e_1. Clearly $\gamma_{i,j,k} \neq 0$ only if $j + k \geqq i$, and if $j + k = i$, then $\gamma_{i,j,k} = 2^{-i}$. We have the following recurrence:

$$p_{m+1,i} = \sum_{\substack{0 \leqq j,k \leqq q \\ j+k \geqq i}} \gamma_{i,j,k} p_{m,j} p_{m,k}.$$

If we now replace each occurrence of $p_{m,0}$ by

$$1 - \sum_{i=1}^{q} p_{m,i},$$

we have the following recurrence:

$$\begin{aligned} p_{m+1,i} = & \sum_{\substack{1 \leqq j,k \leqq q \\ j+k \geqq i}} \gamma_{i,j,k} p_{m,j} p_{m,k} + \sum_{j=i}^{q} \gamma_{i,j,0} p_{m,j} \left(1 - \sum_{s=1}^{q} p_{m,s}\right) \\ & + \sum_{k=i}^{q} \gamma_{i,0,k} \left(1 - \sum_{s=1}^{q} p_{m,s}\right) p_{m,k}. \end{aligned} \tag{6}$$

The values of $\alpha_{i,j}$ and $\beta_{i,j,k}$ are now chosen in a natural way. $\alpha_{i,j}$ is the sum of the coefficients of all $p_{m,j}$ terms in the right-hand side of the recurrence in (6), and $\beta_{i,j,k}$ is the sum of all $p_{m,j}p_{m,k}$ or $p_{m,k}p_{m,j}$ terms in the right-hand side of the recurrence in (6). Then, clearly $\alpha_{i,j} = 0$ if $i > j$ since $p_{m,j}$ does not appear in (6). If $i \leqq j$, then $\alpha_{i,j} = \gamma_{i,j,0} + \gamma_{i,0,j} > 0$ and $\alpha_{i,i} = 2^{1-i}$. If $i > j + k$, then $\beta_{i,j,k} = 0$ since $p_{m,j}p_{m,k}$ or $p_{m,k}p_{m,j}$ does not appear in the right-hand side of the recurrence in (6). If $i = j + k$ and $j = k$, then $\beta_{i,j,k} = \gamma_{i,j,k} = 2^{-i}$, whereas if $i = j + k$ and $j < k$, then $\beta_{i,j,k} = \gamma_{i,j,k} + \gamma_{i,k,j} = 2^{1-i}$. □

We now find a recurrence for z_m and prove certain properties about the recurrence.

THEOREM 4. *There exist constants $\omega_{j,k}$ such that*

(1) *For $m = 0, \cdots, \sigma - 1$,*

$$z_{m+1} = z_m - \sum_{\substack{1 \leqq j,k \leqq q \\ j \leqq k}} \omega_{j,k} p_{m,j} p_{m,k}.$$

(2) *If $j + k = q + 1$, then $\omega_{j,k} = 2^{-q-1}$ if $j = k$ and $\omega_{j,k} = 2^{-q}$ if $j < k$.*

(3) *If $j + k \leqq q$, then $\omega_{j,k} = 0$.*

Proof. For $e \in E_m$, $m > 0$, let Y_e be the number of packets that have traversed e_0 and e_1 and attempt to traverse e (some of these packets may have to be discarded).

Let $p'_{m,i} = \Pr\{Y_e = i\}$. Since packets that traverse e_0 and e_1 will traverse e with probability $\frac{1}{2}$,

$$E(Y_e \mid X_{e_0}, X_{e_1}) = \tfrac{1}{2}(X_{e_0} + X_{e_1}),$$

and, therefore,

$$\begin{aligned}\sum_{i=1}^{2q} ip'_{m+1,i} &= E(Y_e)\\ &= E(E(Y_e \mid X_{e_0}, X_{e_1}))\\ &= E(\tfrac{1}{2}(X_{e_0} + X_{e_1}))\\ &= \sum_{i=1}^{q} ip_{m,i},\end{aligned}$$

and thus

$$\begin{aligned}\sum_{i=1}^{q} ip_{m+1,i} &= \sum_{i=1}^{q} ip'_{m+1,i} + \sum_{i=q+1}^{2q} qp'_{m+1,i}\\ &= \sum_{i=1}^{q} ip_{m,i} - \sum_{i=q+1}^{2q} ip'_{m+1,i} + \sum_{i=q+1}^{2q} qp'_{m+1,i}\\ &= \sum_{i=1}^{q} ip_{m,i} - \sum_{i=q+1}^{2q} (i-q)p'_{m+1,i}.\end{aligned}$$

Since for $i = q+1, \cdots, 2q$,

$$p'_{m+1,i} = \sum_{\substack{1 \leq j,k \leq q\\ j+k \geq i}} \binom{j+k}{i} 2^{-j-k} p_{m,j} p_{m,k}.$$

The theorem follows. □

Suppose that ϕ is a monomial in the variables $w_1, \cdots, w_q$. We define the order of ϕ to be the degree of $\phi(w_1^1, w_2^2, \cdots, w_q^q)$. Let U be the set of all monomials ϕ with degree greater than or equal to two with coefficient equal to one. We define an enumeration $\phi_0, \phi_1, \cdots$, of all monomials in U as follows. Let $\phi_0, \cdots, \phi_{s_1}$ be an enumeration of all monomials in U with degree two such that for $s = 0, \cdots, s_1 - 1$, the order of ϕ_s is less than or equal to the order of ϕ_{s+1}. Let $\phi_{s_1+1}, \cdots, \phi_{s_2}$ be an enumeration of all monomials in U with degree three such that for $s = s_1 + 1, \cdots, s_2 - 1$, the order of ϕ_s is less than or equal to the order of ϕ_{s+1}. Continue to define $\phi_{s_2+1}, \phi_{s_2+2}, \cdots$ in a similar manner. For $s = 0, 1, \cdots$, and $m = 0, 1, \cdots$, let $\phi_{s,m} = \phi_s(p_{m,1}, \cdots, p_{m,q})$. Choose s^* and s^{**} so that $\phi_{s^*} = w_1^{q+1}$ and $\sigma_{s^{**}} = w_1^{q+2}$. Since for $i = 2, 3, \cdots, w_1^i$ is the monomial in U with the smallest order of all monomials in U with degree i, $\phi_0, \cdots, \phi_{s^*-1}$ is an enumeration of all monomials in U with degree less than or equal to q, whereas $\phi_{s^*}, \cdots, \phi_{s^{**}-1}$ is an enumeration of all monomials in U with degree $q+1$.

8. Properties of $\phi_{s,m}$. We now state and prove essential properties of the polynomial $\mathscr{P}$ discussed previously.

THEOREM 5. *There exist constants $d_{s,t}$ such that*

(1) *For $s = 0, 1, \cdots$ and $m = 0, \cdots, \sigma - 1$,*

$$\phi_{s,m+1} = \sum_{t=s}^{\infty} d_{s,t} \phi_{t,m}.$$

(2) *For $s = 0, 1, \cdots, \{t\colon d_{s,t} \neq 0\}$ is finite.*

(3) *If $s<t$ and the order of ϕ_s is greater than the order of ϕ_t, then $d_{s,t}=0$.*

(4) *If $s<t$ and the order of ϕ_s equals the order of ϕ_t, then $d_{s,t}\geqq 0$.*

(5) *If $\phi_s \neq w_1^i$, $i=2,3,\cdots$, then $0<d_{s,s}<1$.*

(6) *If $\phi_s \neq w_1^i$, $i=2,3,\cdots$, then there exists a t such that $s<t$, the order of ϕ_s equals the order of ϕ_t, and $d_{s,t}>0$.*

Proof. We choose $d_{s,t}$ in a natural way. Replace each $p_{m+1,i}$ in $\phi_{s,m+1}$ by the right side of the recurrence in Theorem 3 (1) to yield the following expression:

$$\prod_{i=1}^{q}\left(\sum_{j=1}^{q}\alpha_{i,j}p_{m,j}+\sum_{\substack{1\leqq j,k\leqq q\\ j\leqq k}}\beta_{i,j,k}p_{m,j}p_{m,k}\right)^{\rho_i}, \tag{7}$$

where ρ_i is the exponent of w_i in ϕ_s. Now multiply and simplify the expression in (7) according to the rules of polynomial multiplication and addition, multiplying and adding the $p_{m,i}$ as if they were indeterminates. Then $d_{s,t}$ will be the coefficient of $\phi_{t,m}$ in the resulting expression. Theorem 5 (2) follows since the expression in (7) is formed by multiplying together a finite number of finite terms. Each $d_{s,t}$ will be the sum of terms of the form

$$\prod_{i=1}^{q}\prod_{\rho=1}^{\rho_i}\xi_{i,\rho}, \tag{8}$$

where $\xi_{i,\rho}$ results from replacing a $p_{m+1,i}$ in $\phi_{s,m+1}$ by one of the terms on the right-hand side of the recurrence in Theorem 3 (1), and thus

$$\xi_{i,\rho}\in\{\alpha_{i,j}:1\leqq j\leqq q\}\cup\{\beta_{i,j,k}:1\leqq j,\,k\leqq q,\,j\leqq k\}.$$

Since $\alpha_{i,j}\neq 0$ only if $i\leqq j$ and $\beta_{i,j,k}\neq 0$ only if $j+k\geqq i$, (8) will be nonzero only if $\xi_{i,\rho}$ results from replacing $p_{m+1,i}$ by either $\alpha_{i,j}p_{m,j}$, where $i\leqq j$, or $\beta_{i,j,k}p_{m,j}p_{m,k}$, where $j+k\geqq i$, and thus (8) will be nonzero only if both of the following conditions hold:

- degree of $\phi_t \geqq$ degree of ϕ_s,
- order of $\phi_t \geqq$ order of ϕ_s.

Theorem 5 (3) follows. Furthermore, the only time that (8) will be nonzero when the degree of ϕ_s equals the degree of ϕ_t and the order of ϕ_s equals the order of ϕ_t is when $s=t$. Thus $d_{s,t}$ will be nonzero only if $s\leqq t$, and 5 (1) follows.

If $s<t$ and the order of ϕ_s equals the order of ϕ_t, then for all nonzero terms of the form (8) that contribute to $d_{s,t}$, $\xi_{i,\rho}$ must be either $\alpha_{i,i}$, or $\beta_{i,j,k}$, where $j+k=i$. Since for any of these choices $\xi_{i,\rho}\geqq 0$, Theorem 5 (4) follows.

When $t=s$ there is only one term of the form (8) that contributes to $d_{s,t}$, and in this term each $\xi_{i,\rho}$ equals $\alpha_{i,i}$; thus

$$d_{s,s}=\prod_{i=1}^{q}\alpha_{i,i}^{\rho_i}.$$

Since $\alpha_{i,i}=2^{1-i}$, Theorem 5 (5) follows.

If $\phi_{s,s}\neq w_1^i$, $i=2,3,\cdots$, then there exists an $l\geqq 2$ such that $\rho_l\geqq 1$. Choose t so that ϕ_t is the monomial in U that results from replacing a w_l in ϕ_s by w_1w_{l-1}. Since the degree of ϕ_t is greater than the degree of ϕ_s, $s<t$. Since the order of ϕ_t equals the order of ϕ_s, we know that all terms of the form (8) contributing to $d_{s,t}$ are nonnegative. Furthermore, for one of the terms of the form (8) contributing to $d_{s,t}$, $\xi_{l,1}=\beta_{l,1,l-1}$, and all other $\xi_{i,\rho}=\alpha_{i,i}$, and thus the expression in (8) is greater than zero, and thus $d_{s,t}>0$, proving Theorem 5 (6). □

9. Derivation of modified recurrences. For ease in stating theorems, we will also denote z_m by $z_{m,0}$. For $1\leqq s\leqq s^*$, we now define $z_{m,s}$ and derive a recurrence for $z_{m,s}$

such that the coefficients of $\phi_0, \cdots, \phi_{s-1}$ have coefficients equal to zero and all monomials of degree less than or equal to q also have coefficients equal to zero.

THEOREM 6. *For $s=0, \cdots, s^*$, there exists a polynomial P_s, constants $c_{s,t}$, $t=s, s+1, \cdots$, and $z_{m,s}$, $m=0, \cdots, \sigma$ such that*

(1) *P_s has no nonzero constant or linear terms and*

(2)
$$z_{m,s} = z_m + P_s(p_{m,1}, \cdots, p_{m,q}),$$
$$z_{m+1,s} = z_{m,s} + \sum_{t=s}^{\infty} c_{s,t}\phi_{t,m}.$$

(3) *$\{t : c_{s,t} \neq 0\}$ is finite.*

(4) *If $s \leqq t$ and the order of ϕ_t is less than or equal to q then $c_{s,t}=0$.*

(5) *For $s=1, \cdots, s^*$, either $c_{s-1,s-1}=0$ and $c_{s,t}=c_{s-1,t}$ or $c_{s-1,s-1} \neq 0$ and*
$$c_{s,t} = c_{s-1,t} + \frac{c_{s-1,s-1}d_{s-1,t}}{1-d_{s-1,s-1}}.$$

Proof. We prove the theorem by induction on s. The proof for $s=0$ follows from Theorem 4.

So now suppose that Theorem 6 (1)-(5) is true for s. We prove that Theorem 6 (1)-(5) is true for $s+1$.

If $c_{s,s}=0$, then let $P_{s+1}=P_s$, let $c_{s+1,t}=c_{s,t}$, and let $z_{m,s+1}=z_{m,s}$. Then Theorem 6 (1)-(5) follows immediately.

If $c_{s,s} \neq 0$, then we first observe that by Theorem 6 (4) (using the inductive hypothesis) that the order of ϕ_s is greater than or equal to $q+1$; therefore, $\phi_s \neq w_1^i$, $i=2, \cdots, q$, and thus by Theorem 5 (5), $0 < d_{s,s} < 1$. Now let

$$P_{s+1} = P_s + \frac{c_{s,s}}{1-d_{s,s}}\phi_s,$$
$$z_{m,s+1} = z_m + P_{s+1}(p_{m,1}, \cdots, p_{m,q}), \tag{9}$$
$$c_{s+1,t} = c_{s,t} + \frac{c_{s,s}d_{s,t}}{1-d_{s,s}}.$$

Theorem 6 (1) and (5) now follow.

Theorem 6 (2) follows since

$$\begin{aligned}
z_{m+1,s+1} &= z_{m+1,s} + \frac{c_{s,s}}{1-d_{s,s}}\phi_{s,m+1} \\
&= z_{m,s} + \sum_{t=s}^{\infty} c_{s,t}\phi_{t,m} + \frac{c_{s,s}}{1-d_{s,s}}\sum_{t=s}^{\infty} d_{s,t}\phi_{t,m} \\
&= z_{m,s} + \left(c_{s,s} + \frac{c_{s,s}d_{s,s}}{1-d_{s,s}}\right)\phi_{s,m} + \sum_{t=s+1}^{\infty}\left(c_{s,t} + \frac{c_{s,s}d_{s,t}}{1-d_{s,s}}\right)\phi_{t,m} \\
&= z_{m,s} + \frac{c_{s,s}}{1-d_{s,s}}\phi_{s,m} + \sum_{t=s+1}^{\infty} c_{s+1,t}\phi_{t,m} \\
&= z_{m,s+1} + \sum_{t=s+1}^{\infty} c_{s+1,t}\phi_{t,m},
\end{aligned}$$

which proves Theorem 6 (2).

We know by the inductive hypothesis that $\{t : c_{s,t} \neq 0\}$ is finite. Theorem 6 (3) now follows from (9) and Theorem 5 (2).

To prove Theorem 6 (4) suppose that $t \geqq s+1$ and the order of ϕ_t is less than or equal to q. We have previously observed that our assumption that $c_{s,s} \neq 0$ implies that the order of ϕ_s is greater than or equal to $q+1$. Then by Theorem 5 (3), $d_{s,t}=0$. Since by the inductive hypothesis, $c_{s,t}=0$, it follows from (9) that $c_{s+1,t}=c_{s,t}=0$. □

We now prove that the coefficient of w_1^{q+1} in the recurrence for z_{m,s^*} is negative.

THEOREM 7. $c_{s^*,s^*}<0$.

Proof. We first prove by induction that for $s=0,\cdots,s^*$, the following hold.

(a) If the order of ϕ_t equals $q+1$, then $c_{s,t} \leqq 0$.

(b) There exists a t such that the order of ϕ_t equals $q+1$ and $c_{s,t}<0$.

For $s=0$, (a) and (b) follow from Theorem 4. So now suppose that (a) and (b) are true for s; we prove that they are true for $s+1$. If $c_{s,s}=0$, then (a) and (b) follow immediately by the inductive hypothesis and Theorem 6 (5). If $c_{s,s} \neq 0$, then by Theorem 6 (5),

$$c_{s+1,t}=c_{s,t}+\frac{c_{s,s}d_{s,t}}{1-d_{s,s}}. \tag{10}$$

Since $c_{s,s} \neq 0$ it follows from Theorem 6 (4) that the order of ϕ_s is greater than or equal to $q+1$. Thus $\phi_s \neq w_1^i$, $i=2,3,\cdots$, and, therefore, it follows from Theorem 5 (5) that $0<d_{s,s}<1$.

To prove (a) we consider two cases.

Case 1. The order of ϕ_s is greater than or equal to $q+2$. By the inductive hypothesis, $c_{s,t} \leqq 0$. It follows from Theorem 5 (3) that $d_{s,t}=0$. It now follows from (10) that $c_{s+1,t} \leqq 0$.

Case 2. The order of $\phi_s=q+1$. By the inductive hypothesis we know that $c_{s,s}, c_{s,t} \leqq 0$. By Theorem 5 (4), $d_{s,t} \geqq 0$. It follows from (10) that $c_{s+1,t} \leqq 0$.

To prove (b), we first observe that by the inductive hypothesis there exists a t^* such that the order of ϕ_{t^*} equals $q+1$ and $c_{s,t^*}<0$. We consider two cases.

Case 1. $t^* \neq s$. The proof is similar to the proof of (a).

Case 2. $t^*=s$. Since we have previously observed that $\phi_s \neq w_1^i$, $i=2,3,\cdots$, it follows by Theorem 5 (6) that there exists a t such that $t>s$, the order of $\phi_t=q+1$ and $d_{s,t}>0$. Now we know by (a) (using the inductive hypothesis) that $c_{s,t} \leqq 0$. We have previously assumed that $c_{s,s} \neq 0$, and since the order of $\phi_s=\phi_{t^*}$ equals $q+1$, $c_{s,s}<0$ by (a) (using the inductive hypothesis). We have previously observed that $0<d_{s,s}<1$. It follows from (10) that $c_{s+1,t}<0$.

Since if $s^*<t$ the order of ϕ_t is greater than or equal to $q+2$, it follows from (b) that $c_{s^*,s^*}<0$. □

For $s^*+1 \leqq s \leqq s^{**}$ we now define $z_{m,s}$ and derive a recurrence for $z_{m,s}$ such that monomials of degree less than or equal to q have coefficients equal to zero and the coefficients of $\phi_{s^*},\cdots,\phi_{s-1}$ are negative.

THEOREM 8. *For $s=s^*+1,\cdots,s^{**}$, there exists a polynomial P_s, constants $c_{s,t}$, $t=s,s+1,\cdots$, $z_{m,s}$, $m=0,\cdots,\sigma$, and for $s=s^*,\cdots,s^{**}-1$ there exist constants h_s such that*

(1) *P_s has no nonzero constant or linear terms, and*

$$z_{m,s}=z_m+P_s(p_{m,1},\cdots,p_{m,q}).$$

(2)

$$z_{m+1,s}=z_{m,s}+\sum_{t=s^*}^{s-1} h_t\phi_{t,m}+\sum_{t=s}^{\infty} c_{s,t}\phi_{t,m}.$$

(3) *$\{t: c_{s,t} \neq 0\}$ is finite.*

(4) *$h_t<0$.*

Proof. We prove the theorem by induction on s. The proof for $s=s^*+1$ follows from Theorems 6 and 7.

So now suppose that Theorem 8 (1)-(4) is true for s. We prove that Theorem 8 (1)-(4) is true for $s+1$.

Since for $s^*<s<s^{**}$, ϕ_s has degree $q+1$ and $\phi_s \neq w_1^{q+1}$, it follows by Theorem 5 (5) that $0<d_{s,s}<1$. Now let

$$\begin{aligned} P_{s+1} &= P_s + \frac{1+c_{s,s}}{1-d_{s,s}}\phi_s, \\ z_{m,s+1} &= z_m + P_{s+1}(p_{m,1}, \cdots, p_{m,q}), \\ c_{s+1,t} &= c_{s,t} + \frac{(c_{s,s}+1)d_{s,t}}{1-d_{s,s}}, \\ h_{s+1} &= -1. \end{aligned} \tag{11}$$

Theorem 8 (1) and (4) now follow.

Theorem 8 (2) follows since

$$\begin{aligned} z_{m+1,s+1} &= z_{m+1,s} + \frac{1+c_{s,s}}{1-d_{s,s}}\phi_{s,m+1} \\ &= z_{m,s} + \sum_{t=s^*}^{s-1} h_t\phi_{t,m} + \sum_{t=s}^{\infty} c_{s,t}\phi_{t,m} + \frac{1+c_{s,s}}{1-d_{s,s}}\sum_{t=s}^{\infty} d_{s,t}\phi_{t,m} \\ &= z_{m,s} + \sum_{t=s^*}^{s-1} h_t\phi_{t,m} + \left(c_{s,s} + \frac{(1+c_{s,s})d_{s,s}}{1-d_{s,s}}\right)\phi_{s,m} + \sum_{t=s+1}^{\infty}\left(c_{s,t} + \frac{(1+c_{s,s})d_{s,t}}{1-d_{s,s}}\right)\phi_{t,m} \\ &= z_{m,s} + \sum_{t=s^*}^{s-1} h_t\phi_{t,m} + \left(\frac{1+c_{s,s}}{1-d_{s,s}} - 1\right)\phi_{s,m} + \sum_{t=s+1}^{\infty} c_{s+1,t}\phi_{t,m} \\ &= z_{m,s} + \left(\frac{1+c_{s,s}}{1-d_{s,s}}\right)\phi_{s,m} + \sum_{t=s^*}^{s} h_t\phi_{t,m} + \sum_{t=s+1}^{\infty} c_{s+1,t}\phi_{t,m} \\ &= z_{m,s+1} + \sum_{t=s^*}^{s} h_t\phi_{t,m} + \sum_{t=s+1}^{\infty} c_{s+1,t}\phi_{t,m}, \end{aligned}$$

which proves Theorem 8 (2).

We know by the inductive hypothesis that $\{t: c_{s,t}\neq 0\}$ is finite. Theorem 6 (3) now follows from (11) and Theorem 5 (2). □

10. Proof that $z\to\infty$. In order to invoke Theorem 2 it is now necessary to prove that $z_m\to 0$ as $m\to\infty$. We will prove this by assuming that z_m is bounded below by some positive number, and then by showing that there will be on average enough packets discarded to force z_m below the assumed lower bound.

THEOREM 9. $\lim_{m\to\infty} z_m = 0$.

Proof. Since messages are never created when using the routing scheme it follows that $S_{m+1}\leqq S_m$, and thus $E(S_{m+1})\leqq E(S_m)$. Since $E(S_m)=Nz_m$, $z_{m+1}\leqq z_m$. Since $z_m\geqq 0$, there exists an $L\geqq 0$ such that $\lim_{m\to\infty} z_m = L$. Suppose $L>0$. Then for all $\varepsilon>0$ there exists M such that for $m\geqq M$, $L\leqq z_m<L+\varepsilon$. Choose positive integers η and τ such that $\eta\geqq(q+1)/L$ and $\eta=2^\tau$.

We now show how to partition the switching nodes of the butterfly of size N in levels $M+1$ through $M+\tau$ so that the subgraph induced by each set of nodes in the

partition is isomorphic to the subgraph induced by the set of switching nodes of a butterfly of size η. For $k=0,\cdots,N/\eta-1$, let $b_{\sigma-\tau}\cdots b_1$ be the binary representation of k, and let F_k be the set of all integers with binary representation

$$b_{\sigma-\tau}\cdots b_{M+1}b'_{\tau-1}\cdots b'_1 b_M\cdots b_1,$$

where for $i=1,\cdots,\tau-1$, $b'_i\in\{0,1\}$. Let

$$B_k=\{(l,m): l\in F_k, m=M+1,\cdots M+\tau\}.$$

Then the subgraph of the butterfly graph of size N induced by B_k is isomorphic to the subgraph of the butterfly graph of size η induced by its switching nodes. Let B'_k be the set of edges in E_M that are incident to vertices in B_k, and let B''_k be the set of edges in $E_{M+\tau}$ that are incident to vertices in B_k. Then $\{B'_k\}$ is a partition of E_m and $\{B''_k\}$ is a partition of $E_{M+\tau}$. Let

$$T_k=\sum_{e\in B'_k}X_e,\qquad T^*_k=\sum_{e\in B''_k}X_e.$$

Since

$$E(T_k)=\eta z_M\geqq \eta L\geqq q+1,$$

it follows that $\Pr\{T_k\geqq q+1\}=\lambda>0$. Let Q be $\{k: T_k\geqq q+1\}$. Since $E(|Q|)=\lambda N/\eta$ and $0\leqq k\leqq N/\eta-1$, there exists κ, μ, such that

$$\Pr\{|Q|\geqq \mu N\}\geqq\kappa.$$

Let A be the event $|Q|>\mu N$, and let $\bar{A}$ be the complement of A. Now if $T_k\geqq q+1$, there exists $\zeta>0$ such that the probability that $q+1$ of the packets that traverse some edge in B'_k will attempt to traverse the same edge in the next τ steps is greater than or equal to ζ, and thus

$$E(T^*_k)=E(E(T^*_k\,|\,T_k))\leqq E(T_k-\zeta).$$

Then

$$\begin{aligned} E(S_{M+\tau}\,|\,A)&=E\left(\sum_{k\in Q}T^*\,|\,A\right)+E\left(\sum_{k\notin Q}T^*_k\,|\,A\right)\\ &\leqq E\left(\sum_{k\in Q}T_k-\zeta\,|\,A\right)+E\left(\sum_{k\notin Q}T_k\,|\,A\right)\\ &=E\left(\sum_{k=0}^{N/\eta-1}T_k\,|\,A\right)-E\left(\sum_{k\in Q}\zeta\,|\,A\right)\\ &\leqq E(S_M\,|\,A)-\zeta\mu N,\end{aligned}$$

and thus

$$\begin{aligned} E(S_{M+\tau})&=E(S_{M+\tau}\,|\,A)\Pr\{A\}+E(S_{M+\tau}\,|\,\bar{A})\Pr\{\bar{A}\}\\ &\leqq E(S_M\,|\,A)\Pr\{A\}-\zeta\mu N\Pr\{A\}+E(S_M\,|\,\bar{A})\Pr\{\bar{A}\}\\ &=E(S_M)-\Pr\{A\}\zeta\mu N\\ &\leqq E(S_M)-\kappa\zeta\mu N.\end{aligned}$$

If $\varepsilon\leqq\kappa\zeta\mu$, then

$$z_{M+\tau}\leqq z_M-\kappa\zeta\mu<L+\varepsilon-\varepsilon=L,$$

contradicting $z_m\geqq L$ for $m\geqq M$. $\square$

11. Conclusion of the proof of the principal theorem.

THEOREM 10. *For constant* $q \geqq 2$,

$$z_{m,s^{**}} = (p^{-q} + \Theta(m))^{-1/q}.$$

Proof. Since $\lim_{m\to\infty} z_m = 0$, it follows that for $i \geqq 1$, $\lim_{m\to\infty} p_{m,i} = 0$, and thus we can conclude that $\lim_{m\to\infty} z_{m,s^{**}} = 0$. Now by Theorem 8,

$$z_{m+1,s^{**}} = z_{m,s^{**}} + \sum_{t=s^*}^{s^{**}-1} h_t \phi_{t,m} + \sum_{t=s^{**}}^{\infty} c_{s^{**},t} \phi_{t,m},$$

where only a finite number of the summands of the last sum on the right-hand side of the last equation are nonzero and where for $t = s^*, \cdots, s^{**}-1$, $h_t < 0$. For $t = s^*, \cdots, s^{**}-1$, the degree of ϕ_t equals $q+1$, and for $t = s^{**}, s^{**}+1, \cdots$, the degree of ϕ_t is greater than $q+2$. Since $z_{m,s^{**}} = \Theta(\max\{p_{m,i} : i = 1, \cdots, q\})$ and $\phi_{s^*}, \cdots, \phi_{s^{**}-1}$ is an enumeration of all monomials of degree $q+1$, it follows that

$$z_{m+1,s^{**}} = z_{m,s^{**}} - \Theta(z_{m,s^{**}}^{q+1}).$$

By Theorem 2 it follows that $z_{m,s^{**}} = (p^{-q} + \Theta(m))^{-1/q}$.

Proof of Theorem 1 *when* $p = o((\log N)^{-1/q})$. Since $z_m = z_{m,s^{**}} + O(z_{m,s^{**}}^2)$, it follows that

$$\begin{aligned} z_m &= (p^{-q} + \Theta(m))^{-1/q} + O((p^{-q} + m)^{-2/q}) \\ &= p(1 + \Theta(p^q m))^{-1/q} + O(p^2) \\ &= p(1 + O(p^q m)) + O(p^2) \\ &= p(1 + O(p^q m + p)). \end{aligned}$$

The theorem follows since $S_\sigma = N z_\sigma$. □

We now assume that $p = \Omega((\log N)^{-1/q})$, so that

$$z_m = z_{m,s^{**}} + O(z_{m,s^{**}}^2) = \Theta(m^{-1/q}).$$

THEOREM 11. *For* $i = 1, \cdots, q$, $p_{m,i} = \Theta(m^{-i/q})$.

Proof. We prove by induction on m that for $i \geqq 1$,

$$p_{m+1,i} = \sum_{j=i+1}^{q} \sum_{s=0}^{m} \alpha_{i,i}^{m-s} \alpha_{i,j} p_{s,j} + \sum_{\substack{j \leqq k, j+k \geqq i \\ 1 \leqq j,k \leqq q}} \sum_{s=0}^{m} \alpha_{i,i}^{m-s} \beta_{i,j,k} p_{s,j} p_{s,k}. \tag{12}$$

We simply assume that (12) is true for m, and then to prove it true for $m+1$ we express $p_{m+2,1}$ as a linear combination of $p_{m+1,j}$'s and $p_{m+1,j}p_{m+1,k}$'s using the appropriate recurrence in Theorem 3 (1); replace the $p_{m+1,i}$ in this expression by the right-hand side of (12), and collect similar terms.

We will prove by induction on l the following assertion:

$$\begin{aligned} p_{m,i} &= \Theta(m^{-i/q}), \qquad i = 1, \cdots, l, \\ p_{m,i} &= O(m^{-l/q}), \qquad i = l+1, \cdots, q. \end{aligned} \tag{13}$$

To prove (13) for $l = 1$, we first observe that since

$$z_m = \sum_{i=0}^{q} i p_{m,i} = \Theta(m^{-1/q}),$$

it follows that $p_{m,i} = O(m^{-1/q})$, $i = 1, \cdots, q$. Then by Theorem 3 (1),

$$p_{m+1,1} = \sum_{j=1}^{q} \alpha_{1,j} p_{m,j} + O(m^{-2/q}).$$

Since

$$\sum_{i=1}^{q} ip_{m,i} = \Theta(m^{-1/q}),$$

it follows that $\max\{p_{m,i}: i=1,\cdots,q\} = \Theta(m^{-1/q})$, and since $\alpha_{1,j}>0, j=1,\cdots,q$, it follows that

$$\sum_{j=1}^{q} \alpha_{1,j} p_{m,j} = \Theta(m^{-1/q}),$$

and thus $p_{m,1} = \Theta(m^{-1/q})$.

Now assume that (13) is true for l. We now prove (13) for $l+1$.

We prove by backwards induction on i that $p_{m,i} = O(m^{-(l+1)/q})$, $i=l+1,\cdots,q$. Since $p_{m,j}p_{m,k} = O(m^{-(l+1)/q})$ for j, k such that $j+k \geqq q$ by (13), it follows from Lemma 2 that

$$\sum_{s=0}^{m} \alpha_{q,q}^{m-s} \beta_{q,j,k} p_{s,j} p_{s,k} = O(m^{-(l+1)/q}).$$

It follows from (12) that $p_{m,q} = O(m^{-(l+1)/q})$. Now suppose that

$$p_{m,i'} = O(m^{-(l+1)/q}), \qquad i' = i+1,\cdots,q. \tag{14}$$

By (13), $p_{m,j}p_{m,k} = O(m^{-(l+1)/q})$ for j, k such that $j+k \geqq i$, and thus by Lemma 2,

$$\sum_{s=0}^{m} \alpha_{i,i}^{m-s} \beta_{i,j,k} p_{s,j} p_{s,k} = O(m^{-(l+1)/q}).$$

It follows from (14) and Lemma 2 that for $j \geqq i+1$,

$$\sum_{s=0}^{m} \alpha_{i,i}^{m-s} \alpha_{i,j} p_{s,j} = O(m^{-(l+1)/q}),$$

and thus by (12), $p_{m,i} = O(m^{-(l+1)/q})$.

Now to prove that $p_{m,l+1} = \Theta(m^{-(l+1)/q})$, we first observe that for $i \geqq l+2$, $\alpha_{l+1,i}>0$ by Theorem 3, and $p_{m,i} = O(m^{-(l+1)/q})$ by what we have just proved. If $j+k \geqq l+2$, then $p_{m,j-1}p_{m,k} = O(m^{-(l+2)/q})$ by (13) and (14). If $j+k=l+1$, then $\beta_{l+1,j,k}>0$ by Theorem 3, and $p_{m,j}p_{m,k} = \Theta(m^{-(l+1)/q})$ by (13) (using the inductive hypothesis). It follows by Lemma 2 and (12) that $p_{m,l+1} = \Theta(m^{-(l+1)/q})$. □

THEOREM 12. *If $q \geqq 2$, then for $i=1,\cdots,q$, $p_{m,i} = z_m^i i! + O(z_m^{i+1})$.*

Proof. We prove the theorem by induction on i. To prove $i=1$ we note that since $p_{m,i} = \Theta(m^{-i/q})$, it follows from

$$z_m = \sum_{i=1}^{q} ip_{m,i}$$

that $p_{m,1} = z_m + O(z_m^2)$. We also observe that it follows from Theorem 4 that $z_{m+1} = z_m + O(z_m^2)$.

Now suppose that for $j=1,\cdots,i$, $p_{m,j} = z_m^j/j! + O(z_m^{j+1})$. It follows from Theorems 3 and 11 and the inductive hypothesis that

$$p_{m+1,i+1} = 2^{-i}p_{m,i+1} + \sum_{j=1}^{i} \frac{2^{-i-1}}{j!(i+1-j)!} z_m^{i+1} + r(m),$$

where $r(m)$ is $O(z_m^{i+2})$. Since $p_{m,i+1}$ is $\Theta(p_{m,1}^{i+1})$, there exists a δ such that

$$p_{m,i+1} \leqq \frac{(1+\delta)}{(i+1)!} z_m^{i+1}.$$

Choose C_1 and C_2 so that

$$z_m^{i+1}-z_{m+1}^{i+1}<C_1 z_m^{i+2}, \qquad z_m^{i+2}-z_{m+1}^{i+2}<C_1 z_m^{i+3}.$$

Choose K so that

$$r(m)+C_1(1+\delta)z_m^{i+2}+2KC_2 z_m^{i+3}<Kz_m^{i+2}.$$

We prove by induction on m that

$$p_{m,i+1}\leqq\frac{(1+2^{1-m}\delta)}{(i+1)!}z_m^{i+1}+2K(1-2^{1-m})z_m^{i+2}. \tag{15}$$

Now (15) holds for $m=1$ by choice of δ. So now suppose that (15) holds for m. Then

$$\begin{aligned}
p_{m+1,i+1}&=2^{-i}p_{m,i+1}+\sum_{j=1}^{i}\frac{2^{-i-1}}{j!(i+1-j)!}z_m^{i+1}+r(m)\\
&=2^{-i}p_{m,i+1}+\frac{(1-2^{-i})}{(i+1)!}z_m^{i+1}+r(m)\\
&\leqq 2^{-i}\frac{(1+2^{1-m}\delta)}{(i+1)!}z_m^{i+1}+2^{1-i}K(1-2^{1-m})z_m^{i+2}+\frac{(1-2^{-i})}{(i+1)!}z_m^{i+1}+r(m)\\
&\leqq\frac{(1+2^{-m}\delta)}{(i+1)!}z_m^{i+1}+K(1-2^{1-m})z_m^{i+2}+r(m)\\
&\leqq\frac{(1+2^{-m}\delta)}{(i+1)!}z_m^{i+1}+K(1-2^{1-m})z_m^{i+2}\\
&\quad+Kz_m^{i+2}-C_1(1+\delta)z_m^{i+2}-2C_2Kz_m^{i+3}\\
&\leqq\frac{(1+2^{-m}\delta)}{(i+1)!}z_m^{i+1}-C_1(1+\delta)z_m^{i+2}+2K(1-2^{-m})z_m^{i+2}-2C_2Kz_m^{i+3}\\
&\leqq\frac{(1+2^{-m}\delta)}{(i+1)!}z_{m+1}^{i+1}+2K(1-2^{-m})z_{m+1}^{i+2}.
\end{aligned}$$

We can similarly prove that there exists a K such that

$$p_{m,i+1}\geqq(1-2^{1-m}\delta)z_m^{i+1}-2K(1-2^{1-m})z_m^{i+2}.$$

It follows that $p_{m,i+1}=(1/i!)z_m^{i+1}+O(z_m^{i+2})$. □

Proof of Theorem 1 *when* $p=\Omega(N(\log N)^{-1/q})$. It follows from Theorems 4 and 12 that

$$\begin{aligned}
z_{m+1}&=z_m-\sum_{k=1}^{q}\frac{2^{-q-1}}{k!(q+1-k)!}z_m^{q+1}-O(z_m^{q+2})\\
&=z_m-\frac{(1-2^{-q})}{(q+1)!}z_m^{q+1}-O(z_m^{q+2}).
\end{aligned}$$

It follows from Theorem 2 that

$$z_m=N(p^{-q}+\chi_q m+O(m^{1-(1/q)}))^{-1/q},$$

where

$$\chi_q=\frac{q(1-2^{-q})}{(q+1)!}.$$

The theorem follows by setting $m=\log N$. □

We now consider additional modifications of the butterfly network.

We first consider the d-ary butterfly network. If N, σ are positive integers such that $N = d^\sigma$, then the definition of a d-ary network is similar to our previous definition of the butterfly network. Source and output nodes are defined and labelled as before. Switching nodes are labelled with ordered pairs of integers (l, m) such that $0 \leqq l \leqq (N-1)/d$ and $1 \leqq m \leqq \sigma$. Let $d_{\sigma-1} \cdots d_1$ be the d-ary representation of l. Then, for a switching node (l, m) with $m < \sigma$ and for $i = 0, \cdots, d-1$, let l_i be the integer with d-ary representation $d_{\sigma-1} \cdots d_{m+1} i d_{m-1} \cdots d_1$. Then there is an edge between (l, m) and $(l_i, m+1)$. For switching nodes with $m = 1$ there are edges connecting (l, m) to input nodes with d-ary representation $d_{\sigma-1} \cdots d_1 i$ for $i = 0, \cdots, d-1$, and for switching nodes with $m = \sigma$ there are edges connecting (l, m) to output nodes with binary representation $d_{\sigma-1} \cdots d_1 i$ for $i = 0, \cdots, d-1$. A q-dilated d-ary butterfly network is defined in the same way as before.

We now define the r-replication of a butterfly network. An r-replicated butterfly network is defined by taking r copies of a butterfly network and identifying corresponding input nodes and also by identifying corresponding output nodes. A replicated butterfly network can also be q-dilated and d-ary. A packet to be routed from an input node to an output node now has r paths it can take.

We analyze bandwidth using the same assumptions we made before; each input node independently decides to send a packet with probability p to an output node, with each output node having an equal probability of being chosen, and packets move through the network in synchronized time steps. If the network is r-replicated, we further assume that each packet to be routed independently chooses one of the r possible paths, with each path having an equal probability of being chosen. We have the following theorem.

THEOREM 13. *For constant q and d, the expected bandwidth for the r-replicated, q-dilated d-ary butterfly is*

$$rN\left(\left(\frac{d-1}{2d}\right)\log N + \left(\frac{p}{r}\right)^{-1} + O(\log\log N)\right)^{-1}$$

when $q = 1$ and is

$$Np(1 + O(p^q \log N + p))$$

when $q \geqq 2$ and $p = o((\log N)^{-1/q})$, and when $q \geqq 2$ and $p = \Omega((\log N)^{-1/q})$, the expected bandwidth is

$$rN\left(\left(\frac{p}{r}\right)^{-q} + \chi_q \log_d N\right)^{-1/q} + O(N(\log N)^{-2/q}),$$

where

$$\chi_q = \frac{q(1-d^{-q})}{(q+1)!}.$$

Proof. The proof is similar to the proof of Theorem 1.

Theorem 13 states that when $p = o((\log N)^{-1/q})$, increasing the values of d or r will not significantly increase the expected bandwidth since most packets are already reaching their destinations. When $p = \Omega((\log N)^{-1/q})$, increasing the values of d and r will increase the expected bandwidth by a constant factor when measured as a function of N.

We have discussed several modifications of butterfly networks: q-dilated butterfly networks, d-ary butterfly networks, and r-replicated butterfly networks. These different types of modifications can be combined. For example, a d-ary butterfly network can also be r-replicated and q-dilated. Decisions about which values of q, r, and d to use when building actual machines will be based on comparisons of cost and performance. When building a machine of a fixed cost trade-offs will have to be made about what values of q, d, and r to use.

Increasing the value of q will increase the amount of wire needed and will also require larger, more expensive switches. Increasing the size of r will require more wire and more switches but the switch size will remain the same. Increasing the value of d will use the same number of wires but will require larger switches.

Our results show that when analyzed using our probabilistic assumptions, that for sufficiently large networks of fixed cost increasing q will result in the greatest increase in bandwidth. Since we know that increasing the values of r and q will only change bandwidth by a constant, whereas increasing q will increase bandwidth by an increasing factor of N and that increasing q, d, and r only changes the cost of networks by a constant, we can conclude that for sufficiently large networks increasing q to the greatest value permitted by cost constraints will result in the greatest bandwidth.

12. Concentration of bandwidth about its mean. We now will show that the probability distribution of the bandwidth is tightly concentrated about its expectation. We will need the following theorem of Shamir and Spencer [11].

THEOREM 14. *Suppose that* $Z_0, \cdots, Z_n$ *are random variables defined on a common sample space such that* Z_0 *is a constant function, for* $m=0, \cdots, n$, $E(Z_0)=E(Z_m)$ *and for* $m=0, \cdots, n-1$, $|Z_{m+1}-Z_m| \leqq 1$. *Then*

$$\Pr\{|Z_n - E(Z_n)| \geqq \lambda\} < 2\, e^{-\lambda^2/2n}.$$

THEOREM 15. *For dilated, replicated, and d-ary butterfly networks*

$$\Pr\{|S_\sigma - E(S_\sigma)| \geqq \lambda\} < 2\, e^{-\lambda^2/2N}.$$

Proof. When the routing strategy was described, there was some ambiguity about how to decide which packets to discard. We now give a rule to decide which message to discard. We discard the packets that have the lowest values for the label of the input nodes at which they originated. The distribution of S_σ is independent of the rule used to decide which messages to destroy, but this choice for the rule makes it easier to prove that the distribution of S_σ is sharply concentrated about its mean.

Let $Z_0 = E(S_\sigma)$. For $m=1, \cdots, N$, let W_m be the random variable that is equal to one when a message sent from input node $m-1$ is able to successfully reach its destination, and is equal to zero otherwise. For $m=1, \cdots, N$, let

$$Z_m = E(S_\sigma \mid W_1, \cdots, W_m),$$

i.e., the conditional expectation of the bandwidth when it is known which packets sent from input nodes $0, \cdots, m-1$ successfully reach their destinations, and Z_N is the bandwidth S_σ. Then

$$\begin{aligned} E(Z_m) &= E(E(S_\sigma \mid W_1, \cdots, W_m)) \\ &= E(S_\sigma) \\ &= E(Z_0). \end{aligned}$$

Then for $m=0, \cdots, N-1$, whether a message sent from l, where $l > m$, reaches its destination is not affected by whether a message sent from l^*, where $l^* \leqq m$, reaches

its destination. Thus $Z_{m+1} - Z_m$ is determined by whether or not a message sent from m reaches its destination, and, therefore, $|Z_{m+1} - Z_m| \leqq 1$. The theorem now follows from Theorem 14. □

REFERENCES

[1] N. G. DE BRUIJN, *Methods of Asymptotic Analysis*, North Holland, Amsterdam, 1958.

[2] *Butterfly products overview*, Tech. Report, BBN Advanced Computers, Inc., October, 1987, 27 (1956), pp. 713–721.

[3] T. F. KNIGHT, *Routing statistics for unqueued Banyan networks*, manuscript.

[4] ———, *Technologies for low latency interconnection switches*, Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, June, 1989, pp. 351–358.

[5] C. P. KRUSKAL AND M. SNIR, *The performance of multistage interconnection networks for multiprocessors*, IEEE Trans. Comput., C-32 (1983), pp. 1091–1098.

[6] M. KUMAR AND J. R. JUMP, *Performance of unbuffered shuffle-exchange networks*, IEEE Trans. Comput., C-35 (1986), pp. 573–578.

[7] D. H. LAWRIE, *Access and alignment of data in an array processor*, IEEE Trans. Comput., C-24 (1975), pp. 1145–1155.

[8] J. K. PATEL, *Performance of processor memory interconnections for multiprocessors*, IEEE Trans. Comput., C-30 (1981), pp. 771–780.

[9] M. C. PEASE, *The indirect binary n-cube, microprocessor array*, IEEE Trans. Comput., C-26 (1977), pp. 458–473.

[10] W. RUDIN, *Principles of Mathematical Analysis*, McGraw-Hill, New York, 1965.

[11] E. SHAMIR AND J. SPENCER, *Sharp concentration of the chromatic number on random graphs* $G_{n,p}$, Combinatorica, 7 (1987), pp. 121–130.

[12] H. J. SIEGEL AND R. J. MCMILLEN, *The multistage cube: A versatile interconnection network*, Computer, 14 (1981), pp. 65–76.

[13] H. J. SIEGEL AND S. D. SMITH, *Study of multistage* SIMD *interconnection networks*, Proc. 5th Annual Symposium on Computer Architecture, April, 1978, pp. 223–229.

[14] C. WU AND T. FENG, *On a class of multistage interconnection networks*, IEEE Trans. Comput., C-29 (1980), pp. 694–702.

SIAM J. COMPUT.
Vol. 21, No. 5, pp. 824–855, October 1992
© 1992 Society for Industrial and Applied Mathematics
002

REPRESENTABILITY OF DESIGN OBJECTS BY ANCESTOR-CONTROLLED HIERARCHICAL SPECIFICATIONS*

LIN YU† AND DANIEL J. ROSENKRANTZ†

Abstract. A simple model, called a VDAG, is proposed for succinctly representing hierarchically specified design data in CAD database systems where there are to be alternate expansions of hierarchical modules. The model uses an ancestor-based expansion scheme to control which instances of submodules are to be placed within each instance of a given module. The approach is aimed at reducing storage space in engineering design database systems and providing a means for designers to specify alternate expansions of a module.

The expressive power of the VDAG model is investigated, and the set of design forests that are VDAG-generable is characterized. It is shown that there are designs whose representation via VDAGs is exponentially more succinct than is possible when expansion is uncontrolled. The problem of determining whether a given design forest is VDAG-generable is shown to be *NP*-complete, even when the height of the forest is bounded. However, it is shown that determining whether a given forest is VDAG-generable and producing such a VDAG if it exists, can be partitioned into a number of simpler subproblems, each of which may not be too computationally difficult in practice. Furthermore, for forests in a special natural class that has broad applicability, a polynomial time algorithm is provided that determines whether a given forest is VDAG-generable, and produces such a VDAG if it exists. However, the paper shows that it is *NP*-hard to produce a minimum-sized such VDAG for forests in this special class, even when the height of the forest is bounded.

Key words. hierarchical modules, databases, design objects, versions, module alternatives, conditional expansion, configuration control

AMS(MOS) subject classifications. 68P15, 68Q25, 68R05

1. Introduction. We investigate a model of hierarchically represented design objects, which accommodates design versions and alternatives by permitting the inclusion of a submodule within a larger module to be conditional on the identity of the ancestors of the larger module. This concept of ancestor-controlled expansion of hierarchical modules is formalized by a simple model called a VDAG. The expressibility of the VDAG model is explored, and the design objects that are directly representable via the VDAG model are characterized. Several computational problems dealing with the representability of objects via this model are considered. The computational complexity of these problems is studied, and several appropriate algorithms are developed.

In many design applications, designs are both specified and represented *hierarchically*, where each design object can contain instances of lower level objects within it. This use of hierarchy expedites the design process, and permits very large design objects to be represented relatively succinctly. The issue of storing designs is complicated by the need for *version control* [2], [3], [8], [10], [13], [15], [18] and *design alternatives* [1], [11], [17]. Version control has to deal with multiple versions of a given design object, with the possibility that these versions differ only slightly. Design alternatives involve multiple designs, with the possibility that a given higher level

* Received by the editors July 10, 1989; accepted for publication (in revised form) August 26, 1991. This research was supported in part by the National Science Foundation under grants DCR86-03184 and CCR88-03278. A preliminary extended abstract of this paper appeared in Proceedings of the Ninth Annual ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 1990, Nashville, Tennessee.

† Department of Computer Science, State University of New York at Albany, Albany, New York 12222.

object contains somewhere within it more than one alternative design for instances of a given lower level object. For a recent survey of version control issues, see [9].

In existing version control systems for software engineering and document generation systems, the differences between two versions are usually described on a line basis [13], [15]. The differences between two given files can be computed by algorithms such as those in [6], [7] and the utility program *diff* in Unix system [16]. With this approach, version differences that are kept track of are *line differences.* Sometimes, the differences being kept track of are database units, such as *record differences* [14]. In [3], a technique is proposed for storing different versions of a text file, based on a model in which each version can be envisioned as an AVL tree, each of whose vertices represents a line of text. The set of trees may share common subtrees, and a data structure is proposed that keeps only one copy of certain common subtrees. From an abstract perspective, the method used in [3] is a particular technique for storing a forest of trees compactly, by storing only one copy of common subtrees. This raises the general issue of how to produce a compact representation of a given forest. For this problem, the kind of ancestor-based control of tree expansion considered here can lead to more compact representations.

In this paper, we consider hierarchically specified design objects, and focus on *module differences* in that the basic granularity of differences that are kept track of are instances of submodules within a higher level module. Using module differences to support design alternatives has emerged as an issue in CAD systems. Several schemes have been incorporated in [17]. For instance, one scheme allows a module to have alternate *bodies* that share a common *interface*, and a *configuration* of a module can be created by specifying which alternative body to use for submodules within that module. A configuration can have a different expansion specification for each instance of the same module type within it. Another mechanism provided in [17] is conditional expansions, which can be based on the values of generic parameters. Conditional expansion involves a test to determine whether given submodule instances should be placed within a given module body. The generic parameters, which are typically involved in such tests, are passed in a top-down manner to a given module, and so represent control passed to the module from its ancestors in the hierarchy. A model is proposed in [1] whereby a module can have alternate implementations (corresponding to bodies in VHDL), which share a common interface. A body can have *instantiations* of submodules, and can be *parameterized* by a specification of which body to use for specified occurrences of submodules. However, the model in [1] does not provide any explicit mechanism to control expansion. Furthermore, in the model of [1] certain kinds of alternatives are not supported conveniently, in the sense that they require the creation of separate implementations, even though they may differ only slightly. For example, suppose we want module A to contain certain submodules when it is used as a submodule of B and to contain some other submodules when it is used as a submodule of *C*. In this case, the model in [1] would require two distinct implementations for module *A*.

In this paper, a *hierarchically specified* set of *modules* is a collection of modules, each of which can have a body. A module body contains *instances* of lower level modules, where some of these instances might only be conditionally included in the body. A given *hierarchically specified design module* can be envisioned as a *design tree.* For example, suppose that within the body of design module Z there are three *submodules*, where two are instances of X and the other is an instance of Y; within the body of each module X there are submodules U and V; within the body of each module Y there are two submodules that are instances of W; and the body of each

W contains two submodules that are instances of T. Figure 1.1 shows module Z at different levels of abstraction, and the tree representation of module Z is shown in Fig. 1.2. Note that module Z includes the vertex Z and all of its descendants in the tree. In the tree, for each arc $a = (u, v)$, there is a function $st(a)$ (where "*st*" standards for *stamp*) that provides information about the occurrence of v as a submodule within u.

In the tree of Fig. 1.2, there are two copies of X, U, V, and W, respectively, and four copies of T. If the tree were to be stored directly, there would be a copy of the design data for each *instance* of a module within Z; e.g., there would be four copies of T. However, in CAD systems, a more succinct representation is usually used for hierarchically specified designs; namely, a *directed acyclic multigraph* (*dag*). The dag representation for module Z is shown in Fig. 1.3. Thus, a hierarchically specified set

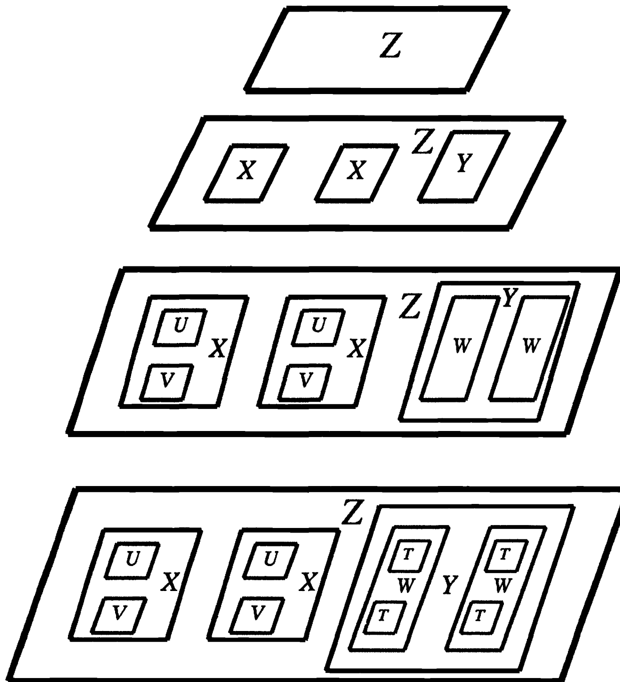


FIG. 1.1

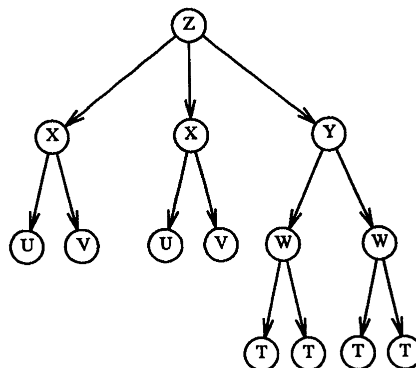


FIG. 1.2

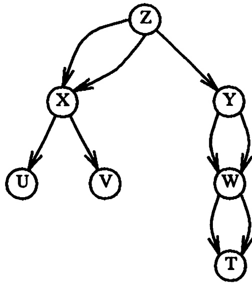


FIG. 1.3

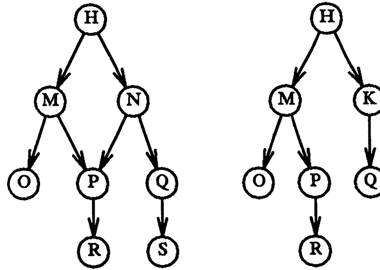


FIG. 1.4

of modules can be represented as a multigraph, where each vertex represents a module. If the body of module Z contains an instance of module X, we say "X is a direct submodule of Z." Corresponding to this instance of X within the body of Z, the dag contains an arc from the vertex representing Z to the vertex representing X. If there is a directed path from Z to X, we say "X is a submodule of Z."

In this dag representation, only one copy of the design data for each module is kept, regardless of the number of the instances of the module involved in the design. The dag representation uses an appropriate stamp to keep track of information about instances of submodules within other modules, such as the two instances of T within W. Since the dag representation reduces duplication of design module descriptions, it is more space efficient to store hierarchically specified design data this way. Also, designers typically use a hierarchical approach to design their modules, so the dag would typically capture the design in the form specified by the designer.

An issue in the formulation of the dag model is that sometimes designers want to have alternative designs for a given module, and sometimes want to use different designs for different instances of a given submodule within a higher level module. We consider the following problem: how to succinctly represent hierarchically specified design module data that supports design alternatives and version control. To illustrate this problem, consider the example shown in Fig. 1.4, where the three multigraphs represent three different versions of a design module H.

The conventional dag representation, as depicted in Fig. 1.3, involves no control over expansion. In the forest represented by such a dag, all the subtrees corresponding to a given vertex of the dag are identical; therefore, expansion control is needed to represent versions and alternatives. The emphasis of this work is on the foundation of mechanisms for controlling expansions. We focus on using the identity of ancestors to control expansion.

Since we do not want to keep multiple copies of the same module for different versions, we can use the following scheme, as illustrated in Fig. 1.5, to store *versioned* hierarchically specified design module data. Under this scheme, we create one source vertex for each design version of module H (note that these newly added source vertices are "dummies" that do not contain actual design data), and we place labels on each arc to indicate to which version or versions the arc belongs. Since the amount of storage for the labels would generally be relatively small compared with the amount of storage saved from eliminating duplicated copies of submodules, this representation is more succinct than a method that keeps all of the design data for each version in separate files. If the number of versions is large and the differences between versions are relatively small, the storage savings can be quite significant.

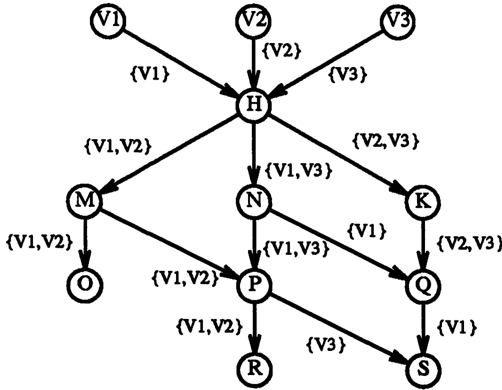


FIG. 1.5

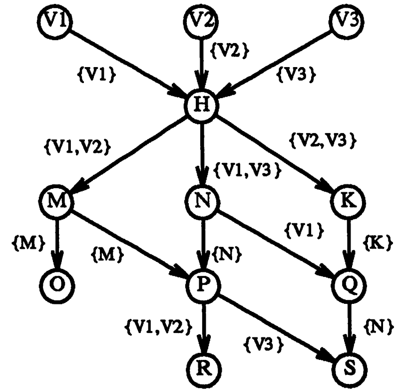


FIG. 1.6

Of course, since we are concerned with succinctness of the design representation, we would like to represent the labels succinctly. For example, in Fig. 1.5, we might use $\{M\}$ as the label for the arc from M to O, instead of $\{V1, V2\}$, since every instance of M in the forest of Fig. 1.4 contains an instance of O. Thus it is appealing to consider a more general scheme in which the elements of a label for an arc from u to v are either u or some ancestors of u in the dag. The example in Fig. 1.6 illustrates how the generalized scheme can be used to represent the three designs of Fig. 1.4.

The VDAG model is a formalization of the technique used in Figs. 1.5 and 1.6, which blends in concerns over version control, design alternatives, and hierarchy by representing hierarchical specifications through alternate expansions of hierarchical modules ("VDAG" stands for *versioned dag*). The VDAG model features an ancestor-based expansion scheme to control which instances of submodules are to be placed within each instance of a given module. The approach is aimed at reducing storage space in engineering design database systems and providing a capability for designers to specify alternate expansions of a module. The VDAG model is not restricted to any specific design applications, such as VLSI design or civil engineering design.

In this formalization, each module is represented by a VDAG vertex having a unique tag. Each possible use of one module is a direct instance within a larger module is represented as a VDAG arc. The arc is labeled with a specification as to when the potential instance should indeed be a real instance within the larger module. This specification is ancestor-based. It can say that the instance should always be included within the larger module, or it can say that the instance should be included only if some member of a given list of modules is an ancestor of the larger module.

Not every design forest can be generated by a VDAG. The issue of which design forests can be represented by VDAGs is explored here. A forest that cannot be generated by a VDAG would have to be modified in order to be VDAG-generable. Such a modification would take the form of changing the identity of some of the vertices in the forest so that they can be represented by distinct vertices in the VDAG. However, a conventional model of hierarchical designs, where all expansions are unconditional, would require more duplication than the VDAG model requires. A typical conventional CAD system might have a file for each module, perhaps with the file name the same as the module name. If a variant of a given module is needed, the file is copied, modified appropriately, and the variant module is renamed. Instances of the given module (within larger modules) that are to use the new variant are modified to use the new module name.

In [19] we introduced the VDAG model, provided some algorithms to process VDAGs, and investigated some combinatorial problems involved in processing a given VDAG.

The remaining sections are organized as follows. In § 2, we present some basic definitions and concepts. In § 3, we investigate the expressive power of the VDAG model. The complexity of determining whether a design forest can be generated by a VDAG is examined in § 4. An important natural class of design forests is identified in § 5, and a polynomial time algorithm is provided to build a VDAG, when one exists, for a given design forest in that class. In § 6, we address the search space issue in construction of arc labels. In § 7, non-VDAG-generable forests are dealt with. The relative conciseness of the VDAG model in comparison with the conventional dag model is examined in § 8. Simplification problems are considered in § 9.

2. Basic definitions and concepts.

DEFINITION. In a dag $G(V, A)$, where V is a set of vertices and A a set of arcs, a vertex u is an *ancestor* of a vertex v if u is v or there is a directed path from u to v in G.

DEFINITION. A *design tree* is a triple (T, t, st) where T is a tree, t is a function that assigns each vertex v of T a value $t(v)$ called the *tag* of v, and st is a function that assigns each arc a of T a value $st(a)$ called the *stamp* of a.

In a design tree, each vertex v represents a *module.* The tag $t(v)$ on a vertex v contains the design data for the module represented by that vertex. We assume that a portion of a tag serves as a module identifier. The information in the tag may also include an interface description describing how the module is connectable when used as a direct submodule within a larger module. For example, the tag might contain a formal parameter list, comparable to a list of input and output ports of a VLSI module.

An arc a from vertex u to vertex v represents an instance of module v as a submodule occurring within module u. For arc a, the stamp $st(a)$ is the information specifying how the instance of v occurs inside u. For example, the stamp may specify the location and/or orientation of the instance within u. The stamp might also contain an actual parameter list; for instance, in the VLSI application $st(a)$ might specify which signal of u is connected to each port of the instance.

Note that a design tree is an unordered tree. If it is desired that the ordering of children of a vertex should have some significance, this ordering information can be incorporated in the stamps on the arcs going to the children. In that case, the arcs existing from the same vertex will have distinct stamps.

DEFINITION. A *design forest* is a set of design trees such that each tag of a root vertex contains an identifier occurring nowhere else in the forest.

The tag on the root of each tree serves to uniquely identify the tree as a design module, or perhaps as a particular version or alternative of a design module.

In the future we often use "tree" and "forest" to mean design tree and design forest, respectively.

DEFINITION. A VDAG is a four tuple (G, t, st, l), where G is a directed, acyclic multigraph with vertex set V and arc set A; t is a function mapping each vertex v to a unique value denoted by $t(v)$, (t for *tag*); st is a function mapping each arc a to a value denoted by $st(a)$, (st for *stamp*); and l is a function mapping each arc a to a nonempty subset of ancestors of the vertex exited by a, and is denoted by $l(a)$ (l for *label*).

Since each VDAG vertex has a tag containing a unique identifier, we assume that a label $l(a)$ for an arc a is represented as a list of tag identifiers. (For convenience in

presenting results and examples, we will often equate a tag with its tag identifier, but in practice we anticipate that a vertex would contain an entire tag, and a label would contain just tag identifiers.) The interpretation of element w in $l(a)$, where arc a goes from u to v, is that whenever an instance of module w has an instance of u as a submodule, or a submodule of a submodule, etc., then each instance of u within the instance of w should contain an instance of v within the instance of u. This concept is formalized below in the definition of a "generating path."

Each vertex of the VDAG might have the format shown in Fig. 2.1, where vertex v has k exiting arcs pointing to (not necessarily distinct) submodules $v_1, \cdots, v_k$. The *connection data* contains stamps, labels, and pointers to submodules. The *design data* contains the tag of v. Fig. 2.1 is only intended to be suggestive of how the design information might be stored, and many variations are possible. For instance, there might be a separate file for each VDAG vertex.

Example. Given a forest of two trees representing two versions of a design as shown in Fig. 2.2(a), a possible VDAG representation is shown in Fig. 2.2(b). (In this example, the arcs all have stamp δ.)

From the examples given above, we observe that if there is a path in the forest that starts at a root vertex with tag A and ends at a vertex with tag B, then module A contains an instance of B as a submodule. Each such path in the forest corresponds to a VDAG path that starts at the vertex with tag A and ends at the vertex with tag B, and has appropriate labels and stamps on its arcs. Similarly, for each properly labeled path in the VDAG, there should be a corresponding path in the design forest. However, care is needed in formalizing the concept of properly labeled paths. For instance, in Fig. 2.2(a) A contains D but B does not, even though there may seem to be a path from B to D in Fig. 2.2(b). To capture the concept of a properly labeled VDAG path, which corresponds to the containment relationship, we formulate the following definition. The idea behind this definition is that a given path can be extended

design data for vertex v $tag(v)$			
connection data for arc a_1: $stamp(a_1)$, $label(a_1)$, *pointer to* v_1	connection data for arc a_2: $stamp(a_2)$, $label(a_2)$, *pointer to* v_2	$\cdots$	connection data for arc a_k: $stamp(a_k)$, $label(a_k)$, *pointer to* v_k

FIG. 2.1

FIG. 2.2

by a given arc exiting the path's endpoint only if the path contains at least one vertex that is an element of the given arc's label.

DEFINITION. In a given VDAG (G, t, st, l), a *valid path* is either a single vertex v_0, or is a sequence of arcs $a_1, a_2, \cdots, a_k$ in A, $k \geqq 1$, such that there exist vertices $v_0, v_1, \cdots, v_k$ for which for all i, $1 \leqq i \leqq k$, arc a_i connects v_{i-1} to v_i, and the intersection of $l(a_i)$ and $\{v_0, v_1, \cdots, v_{i-1}\}$ is nonempty. A *generating path* is a valid path whose initial vertex is a source of G.

Note that, by the definition of a generating path, for each source vertex v in V, there is a generating path from v to v, and if a sequence of arcs $a_1, a_2, \cdots, a_k$ is a generating path, then so are each of its prefixes of the form $a_1, a_2, \cdots, a_j$, where $1 \leqq j < k$. The significance of a generating path from v_0 to v_k is that an instance of module v_0 contains an instance of v_k as a submodule because of that generating path. In particular, v_0 contains an instance of v_1, which in turn contains an instance of v_2, etc. The generating path corresponds to this sequence of nested module instances.

DEFINITION. Given a VDAG β, the *exploded forest generated by* β is a forest F with a distinct vertex for each distinct generating path in β. The tag on the vertex corresponding to such a path is the same as the tag of the last vertex in the path. For each source vertex v_0 in β, forest F contains a root vertex corresponding to the generating path consisting of the single vertex v_0. For each generating path consisting of a single arc a from a source vertex v_0 to a vertex v_1, the tree vertex corresponding to the generating path v_0 is the parent of the tree vertex corresponding to the generating path a. For each generating path $a_1, \cdots, a_k$, having at least two arcs, the tree vertex corresponding to the generating path $a_1, \cdots, a_{k-1}$ is the parent of the tree vertex corresponding to $a_1, \cdots, a_k$. There are no other vertices and arcs in F. The stamp on the tree arc between a parent and a child is the same as the stamp of the last arc in the generating path for the child. Given a VDAG β, we will denote the exploded forest generated by β as F_β. We say that a design forest F is *VDAG-generable* if there exists a VDAG β such that $F = F_\beta$.

Given a VDAG β, the exploded forest generated from the VDAG is a set of design trees, with a tree root for each source vertex in β. The relationship between a given VDAG and this set of design trees constitutes the meaning of the VDAG; the purpose of the VDAG is to represent the set of design trees that it generates.

Recalling Fig. 2.2, the exploded forest generated by the VDAG in Fig. 2.2(b) is the forest shown in Fig. 2.2(a). Figure 2.3 is the exploded forest generated by the VDAG in Fig. 1.6.

We now show that the problem of finding a VDAG that generates a given forest is no harder than the problem of finding a VDAG that generates a given tree. Consider an algorithm SUPERTREE, which given a forest F containing k trees with source tags $s_1, s_2, \cdots, s_k$, as shown in Fig. 2.4(a), returns a forest consisting of a single tree, as follows. (1) Add a "super source" with a tag S not in F; (2) make $s_1, s_2, \cdots, s_k$ each a child of S where the stamps on the arcs from S to its children are assigned arbitrary unique values $\delta_1, \delta_2, \cdots, \delta_k$, as shown in Fig. 2.4(b).

PROPOSITION 2.1. *A forest F is* VDAG-*generable if and only if* SUPERTREE(F) *is* VDAG-*generable.*

Proof. (*if*) Suppose that a VDAG α, as shown in Fig. 2.4(c), generates SUPERTREE(F). A VDAG β generating F can be constructed from α as follows. First vertex S and its exiting arcs are deleted. Then each occurrence of S as a label element in a remaining arc, say from vertex u to vertex v, is replaced by those members of the set $\{s_1, s_2, \cdots, s_k\}$ that are ancestors of u. (VDAG α and β are illustrated in Figs. 2.4(c) and (d), respectively.) It is easy to see that β generates F.

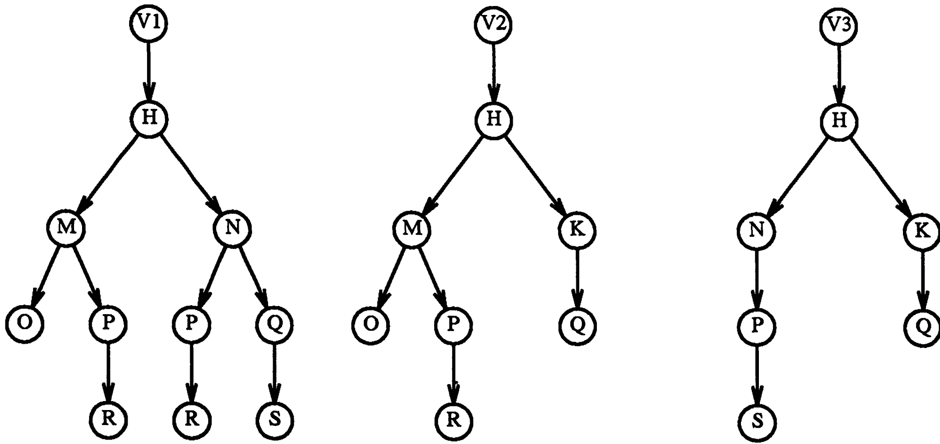


FIG. 2.3

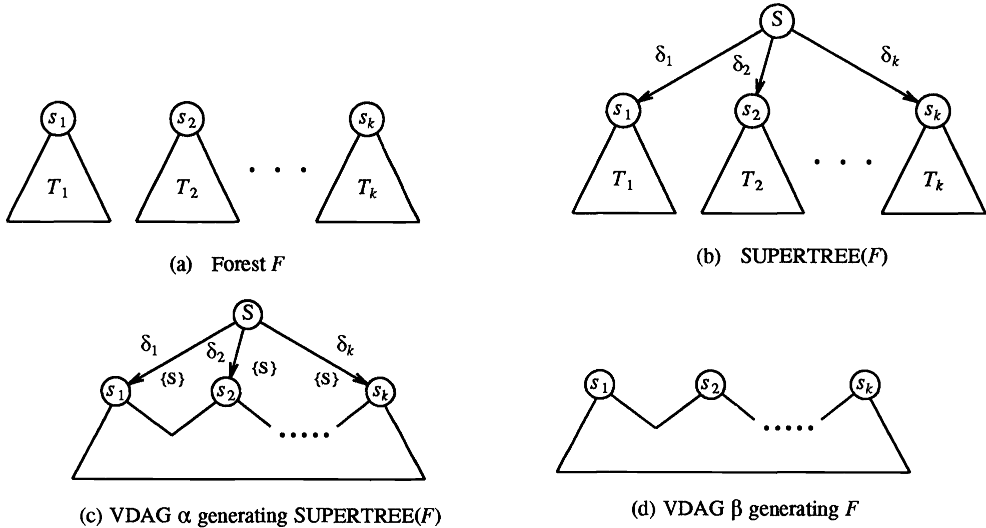


FIG. 2.4

(*only if*) Suppose a VDAG β, as shown in Fig. 2.4(d), generates F. Then SUPERTREE(F) is generated by the VDAG α shown in Fig. 2.4(c), which is obtained from β by adding a new vertex of tag S and for each i, $1 \leqq i \leqq k$, one arc from S to s_i with stamp δ_i and label $\{S\}$. □

From the proof of Proposition 2.1, we observe that the problem of finding a VDAG that generates a given forest is not significantly harder for forests containing multiple trees than for forests containing a single tree.

A VDAG arc, vertex, or label element that is uninvolved in any generating path is useless in producing the exploded forest represented by the VDAG. The set of VDAGs without such useless components can be formalized as follows.

DEFINITION. A VDAG is *valid* if each arc occurs on some generating path, and if each element of each arc label occurs on some generating path that leads to the vertex exited by the arc.

LEMMA 2.2. *Given an invalid* VDAG α, *there exists a valid* VDAG *that generates the same forest as* α *does.*

Proof. Let β be the VDAG obtained from α by deleting vertices and arcs that do not occur on any generating paths, and deleting arc label elements that do not occur on any generating path leading to the vertex exited by the arc. Then β is a valid VDAG that generates the same forest as α. □

3. Expressive power of the VDAG model. In § 1, we pointed out that there are certain forests that are not VDAG-generable. This is a consequence of the requirement that each VDAG vertex has a unique tag, and that in generating the exploded forest represented by a given VDAG, the expansion of a given vertex in the forest can depend only on the ancestors of that vertex.

For example, there are no VDAGs that generate the design forests in Fig. 3.1(a) and Fig. 3.1(b), since each tree contains two instances of B that are indistinguishable by the tags of their ancestors, but which are expanded differently. It is also the case that the VDAG model cannot be used for forests that represent recursively defined design modules. An example of such a forest is shown in Fig. 3.1(c).

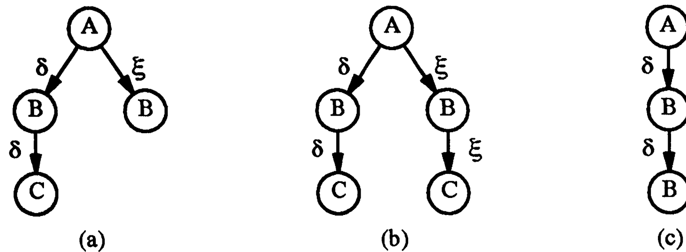


FIG. 3.1

If a given design forest cannot be generated by a VDAG, the forest can be modified by using new tag values for certain vertices. For instance, in each of the examples of Fig. 3.1, one of the B vertices could have its tag changed to B', and the modified design forest would be VDAG-generable. The VDAG would have separate vertices with tags B and B', with replication of data in the tag.

We now investigate the conditions under which a design forest F can be generated by a VDAG.

DEFINITION. Given a vertex u in a forest F, $pathtag(u)$ is the set of all tags of vertices on the path from a forest root to u (including the tag of u itself). For each tag t in F, $anctag(t)$ is the union of $pathtag(u)$ over all vertices having tag t.

Consider a forest F and a given tag t. Consider the set of paths starting at a root and ending at a vertex with tag t. As shown in Fig. 3.2, let these paths be $p_1, p_2, \cdots$, and p_k, having endpoints $u_1, u_2, \cdots, u_k$, respectively. Suppose there is a u_i, $1 \leqq i \leqq k$, having m exiting arcs with stamp δ that enter vertices with tag t', $m \geqq 0$, and there is a u_j, $1 \leqq j \leqq k, j \neq i$, having n exiting arcs with stamp δ that enter vertices with tag t', where $n > m$. Then in each VDAG that generates F, if any, there must be at least n arcs with stamp δ going from t to t'. Furthermore, of this set of arcs, there must be $n-m$ arcs having labels that are each disjoint from $pathtag(u_i)$ but not disjoint from $pathtag(u_j)$. In other words, there must be $n-m$ arcs whose label l satisfies the conditions that $l \cap pathtag(u_i) = \emptyset$ and $l \cap pathtag(u_j) \neq \emptyset$. An obvious necessary condition for this is that $pathtag(u_j) - pathtag(u_i)$ is nonnull. However, the requirements on the VDAG are more subtle.

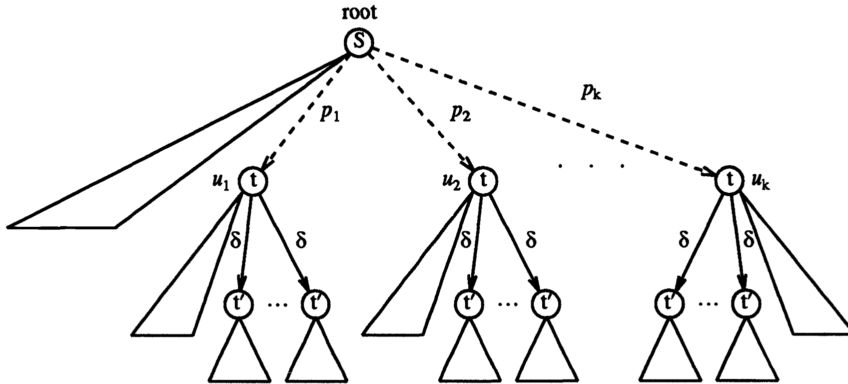


FIG. 3.2

DEFINITION. For a given forest F, tag t, tag t', and stamp δ, if F contains at least one arc with stamp δ going from a vertex with tag t to a vertex with tag t', we say that (t, t', δ) is a *relevant triple* for F. We call a forest or VDAG arc that exits a vertex with tag t, enters a vertex with tag t', and has stamp δ, a *(t, t', δ)-arc.*

It will turn out that whether or not a given forest is VDAG-generable involves a certain combinatorial property for each relevant triple. First consider the following definitions pertaining to relevant triples.

DEFINITION. Given a vertex u with tag t, a tag t', and a stamp δ in a forest F, let $num(u, t', \delta)$ be the number of (t, t', δ)-arcs exiting from u. Given a relevant triple (t, t', δ) for F, let $maxnum(t, t', \delta)$ be the maximum number of (t, t', δ)-arcs exiting from a vertex with tag t, i.e.,

$$maxnum(t, t', \delta) = \max \{num(u, t', \delta) \mid u \text{ has tag } t\},$$

and let $totnum(t, t', \delta)$ be the total number of (t, t', δ)-arcs, i.e.,

$$totnum(t, t', \delta) = \sum_{\text{vertices } u \text{ with } tag\ t} num(u, t', \delta).$$

To capture the labeling requirements imposed by F, we define the concept of a "number function" for a relevant triple, as follows.

DEFINITION. Given a design forest F and a relevant triple (t, t', δ), a *number function* $N_{t,t',\delta}$ for F and (t, t', δ) is a mapping from the nonempty subsets of $anctag(t)$ to the nonnegative integers, i.e.,

$$N_{t,t',\delta}\colon (2^{anctag(t)} - \varnothing) \to \mathbf{N},$$

such that for all vertex u in F with tag t,

$$num(u, t', \delta) = \sum_{\substack{\gamma \text{ such that} \\ \gamma \cap pathtag(u) \neq \varnothing}} N_{t,t',\delta}(\gamma).$$

The intuition behind a number function is that it specifies the number of arcs having each label in a particular VDAG for F; i.e., $N_{t,t',\delta}(\gamma)$ is the number of (t, t', δ)-arcs labeled with γ. One observation on $N_{t,t',\delta}$ is that it need not be unique.

We now define some properties that will characterize the VDAG-generable forests.

DEFINITION. A design forest F is *tag-acyclic* if merging vertices of F with the same tag does not create cycles.

DEFINITION. A relevant triple for a design forest is *labelable* if a number function exists for it. A design forest is *labelable* if all of its relevant triples are labelable.

DEFINITION. A design forest is *well structured* if it is tag-acyclic and labelable.

It will be shown that a given forest is VDAG-generable if and only if it is well structured.

LEMMA 3.1. *Every well-structured design forest is* VDAG-generable.

Proof. Consider a well-structured design forest F. Consider any directed multigraph with a vertex for each tag occurring in F, and a set of arcs satisfying the constraint that there is an arc from the vertex with tag t to the vertex with tag t' only if F contains an arc from a vertex with tag t to a vertex with tag t'. Since F is tag-acyclic, any such multigraph is acyclic.

Now consider the following directed acyclic multigraph α of the above form. Multigraph α has a vertex for each tag occurring in F. For each relevant triple (t, t', δ), let $N_{t,t',\delta}$ be a number function; since F is labelable, such a number function exists. Based on this number function, multigraph α is given a set of labeled (t, t', δ)-arcs. For each set γ such that $N_{t,t',\delta}(\gamma)$ is nonzero, the number of (t, t', δ)-arcs with label γ in α is $N_{t,t',\delta}(\gamma)$. Note that the elements of each such label γ are ancestors of the vertex for t in α; therefore, the labeled directed acyclic multigraph α is a VDAG. The constraints on $N_{t,t',\delta}$ guarantee that the exploded forest generated by α is the original design forest. □

LEMMA 3.2. *If a design forest is* VDAG-*generable, it is well structured.*

Proof. Consider a design forest F. Suppose F is generated by VDAG α. By Lemma 2.2, we may assume that α is valid.

Since α does not contain cycles and each arc of F corresponds to an arc of α whose endpoints have the same tags as the arc in F, F must be tag-acyclic.

The remaining task is to show that F is labelable. We claim that for each relevant triple (t, t', δ) for F, a number function $N_{t,t',\delta}$ exists. For a given tag t, let v_t be the vertex of α whose tag is t. Because α is valid, every label element x occurring in the label of an arc exiting the VDAG vertex v_t has the property that α has a generating path p going from a source vertex to vertex v_t, such that x occurs on p. Since forest F is the exploded forest generated from α, F contains a vertex u corresponding to path p. This vertex u has tag t, and has x as a member of $pathtag(u)$. Consequently, x is a member of $anctag(t)$. Since this is true for each label element of each arc exiting v_t, the label of each arc exiting v_t is a nonempty subset of $anctag(t)$. Now, on the basis of α, define a function $N_{t,t',\delta}$ from $(2^{anctag(t)} - \varnothing)$ to N as follows. For each nonempty subset γ of $anctag(t)$, define $N_{t,t',\delta}(\gamma)$ to be the number of (t, t', δ)-arcs in α with label γ.

We now show that the specified function $N_{t,t',\delta}$ is indeed a number function for F. Consider a vertex u of F having tag t. Since F is the exploded forest generated from α, VDAG α has a generating path p corresponding to u, such that this generating path ends at the vertex v_t whose tag is t. Furthermore, each (t, t', δ)-arc exiting vertex u in F corresponds to an extension of path p to a longer generating path by the use of a VDAG arc having stamp δ that exits v_t and enters the vertex $v_{t'}$ having tag t', such that the arc's label γ has a nonnull intersection with the vertices in path p. Since γ is a subset of $anctag(t)$, this arc contributes to the value of $N_{t,t',\delta}(\gamma)$. Thus $num(u, t', \delta)$ equals the number of (t, t', δ)-arcs in α having a label whose intersection with the vertices on path p is nonempty. Furthermore, each such arc has a label that is a subset of $anctag(t)$. Consequently,

$$\sum_{\substack{\gamma \text{ such that} \\ \gamma \cap pathtag(u) \neq \varnothing}} N_{t,t',\delta}(\gamma) = num(u, t', \delta).$$

Therefore, for each relevant triple (t, t', δ), the specified function $N_{t,t',\delta}$ is indeed a number function for F. Hence F is labelable. □

A consequence of Lemmas 3.1 and 3.2 is the following.

THEOREM 3.3. *A design forest is* VDAG-*generable if and only if it is well structured.*

4. VDAG construction. We now consider the problem of determining if a given design forest F is well structured. First consider tag-acyclicity. To test tag-acyclicity, a given forest can be collapsed into a graph having single vertex for each tag in the forest (thereby merging all forest vertices having the same tag), and having a single arc for all the forest arcs whose endpoints have the same tag. This conversion is described in algorithm COLLAPSE, shown in Fig. 4.1.

PROPOSITION 4.1. *A forest F is tag-acyclic if and only if* COLLAPSE(F) *is acyclic.*

Proof. The proof is obvious. □

An observation on the tag-acyclicity of a forest F is that it can be tested in linear time. The graph COLLAPSE(F) can be constructed in linear time, and a topological sorting on COLLAPSE(F) can be done in time linear in its size [12].

ALGORITHM COLLAPSE

Input: forest $F=(T, t, st)$

Output: a graph

1. for each distinct tag t in F, create a vertex with tag t and call the vertex t;
2. for each pair of tags (t, t') such that in F, a vertex with tag t is a parent of a vertex with tag t', create an edge from t to t' in the graph.

FIG. 4.1

If a given forest F is tag-acyclic, we can perform a labelable test on F to determine whether it is well structured. Detecting the labelable condition, however, can be a difficult computational task, as indicated by the following result. The problem of determining if a forest is VDAG-generable is NP-complete; furthermore, the problem is NP-complete even for forests of bounded height.

DEFINITION. The *height* of a forest F is the number of arcs on a longest path from a source vertex to a leaf vertex. The *depth* of a vertex u in a forest is the number of arcs on the path from a source vertex to u.

We observe that Proposition 2.1 implies that (1) if the VDAG-generability problem for forests of height h is NP-hard, the VDAG-generability problem for trees of height $h+1$ is also NP-hard; and (2) if the VDAG-generability problem for trees of height h is computationally easy, the VDAG-generability problem for forests of height $h-1$ is also easy.

The following result concerns the size of a VDAG with respect to the size of a forest it represents.

THEOREM 4.2. *For a* VDAG-*generable forest F and a number function $N_{t,t',\delta}$ for F, it is the case that*

$$\sum_{\gamma\in(2^{anctag(t)}-\varnothing)} N_{t,t',\delta}(\gamma) \leqq totnum(t, t', \delta).$$

Proof. Consider a number function $N_{t,t',\delta}$ for relevant triple (t, t', δ). For each vertex u with tag t and at least one exiting (t, t', δ)-arc, it is the case that

$$num(u, t', \delta) = \sum_{\substack{\gamma \text{ such that} \\ \gamma\cap pathtag(u)\neq\varnothing}} N_{t,t',\delta}(\gamma).$$

Because of the above equality, it is possible to *associate* each of the forest arcs that contribute to $num(u, t', \delta)$ with a set γ having a nonnull intersection with $pathtag(u)$, such that exactly $N_{t,t',\delta}(\gamma)$ of these arcs are associated with each set γ. Since the association can be done for each vertex u having tag t, each arc that contributes to $totnum(t, t', \delta)$ is associated with exactly one set γ.

Now consider a set γ such that $N_{t,t',\delta}(\gamma)$ has a nonzero value, say k. Because γ is a nonnull subset of $anctag(t)$, F contains a vertex u with tag t such that $\gamma \cap pathtag(u)$ is nonnull. Consequently, there are k arcs exiting u that have been associated with γ. Since this is true for each γ, and the total number of arcs available for association is only $totnum(t, t', \delta)$, it is the case that

$$\sum_{\gamma \in (2^{anctag(t)} - \varnothing)} N_{t,t',\delta}(\gamma) \leqq totnum(t, t', \delta). \qquad \square$$

THEOREM 4.3. *For each $h \geqq 4$, given a design tree F of height h that is tag-acyclic, the problem of determining if F is* VDAG-*generable is NP-complete.*

Proof. Theorem 3.3 implies that F has a VDAG representation if and only if a labeling can be found. By Theorem 4.2, for each relevant triple (t, t', δ), the number of sets γ such that $N_{t,t',\delta}(\gamma)$ is nonzero cannot exceed the number of vertices in F. Thus a nondeterministic Turing machine can guess a labeling and verify in time polynomial in the size of F that the labeling satisfies the constraints of the labelable condition. Consequently, the set of well-structured forests is in *NP*.

The *NP*-hardness is by a reduction from the Graph 3-Coloring Problem (for undirected graphs) [4].

Consider graph $G(V, E)$, where $V = \{v_1, v_2, \cdots, v_n\}$, which is to be 3-colored. We construct a design tree F as follows. The root of F, denoted *root*, has tag r. For each v_i in V, F contains four distinct tags, v_i^1, v_i^2, a_i^1, and a_i^2. Also, for each v_i in V, *root* has two subtrees, headed by v_i^1 and a_i^1, respectively, as shown in Fig. 4.2. For each edge (v_i, v_j) in V, $i < j$, the vertex with tag v_i^1 previously placed in F has a subtree, headed by v_j^2, as shown in Fig. 4.3. F also contains three distinct tags b^1, b^2, and Q, where *root* has two subtrees, headed by Q and b^1, respectively, as shown in Fig. 4.4. All arcs in F have stamp δ.

For instance, given the Graph 3-Coloring instance shown in Fig. 4.5, the constructed forest F is shown in Fig. 4.6. A VDAG generating F is shown in Fig. 4.7.

Note that the size of F is linear in the size of the given graph, that F has height 4, and that F is tag-acyclic. Also, as illustrated in Fig. 4.7, we observe that all arcs, except for (t, t', δ)-arcs, can be labeled by $\{r\}$; so there is clearly a number function for each relevant triple, with the possible exception of the triple (t, t', δ).

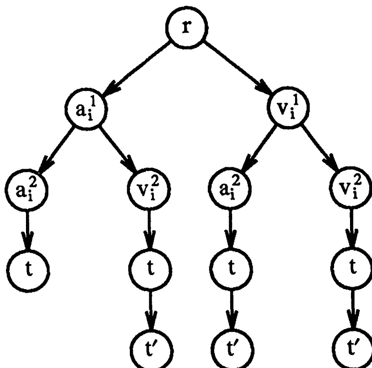


FIG. 4.2

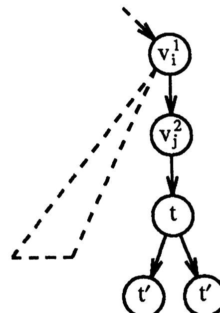


FIG. 4.3

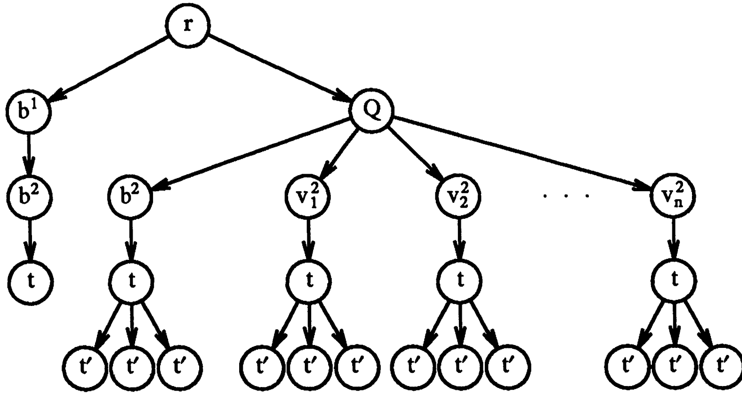


FIG. 4.4

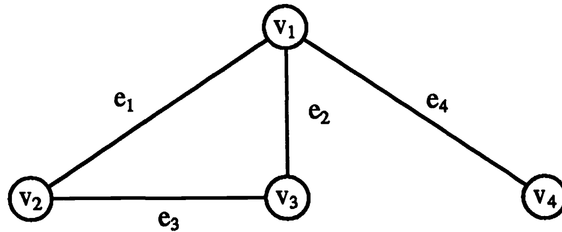


FIG. 4.5

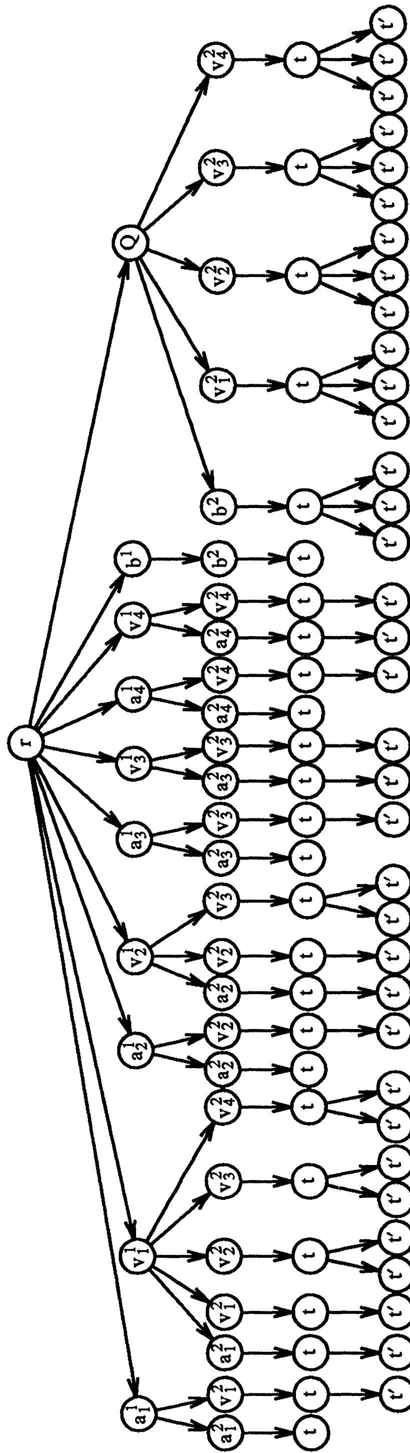
Suppose F can be generated by a VDAG. Consider VDAG arcs from t to t'. Since the forest paths r, b^1, b^2, t and r, a_i^1, a_i^2, t, for $1 \leqq i \leqq n$, end in vertices with no children, the VDAG cannot contain any (t, t', δ)-arc whose label contains r, t, b^1, b^2, a_i^1, or a_i^2.

For each $i, 1 \leqq i \leqq n$, the paths r, a_i^1, v_i^2, t and r, v_i^1, a_i^2, t require that v_i^1 and v_i^2 each must appear in exactly one (t, t', δ)-arc label, since each of the occurrences of t on the two paths has only one child whose tag is t'. Because the occurrence of t in path r, v_i^1, v_i^2, t has exactly one child of tag t', tags v_i^1 and v_i^2 must appear in the same (t, t', δ)-arc label.

Consider the path r, Q, b^2, t. Since this t has 3 children whose tag is t', the VDAG must contain exactly three (t, t', δ)-arcs whose labels contain Q. For each $i, 1 \leqq i \leqq n$, because of the path r, Q, v_i^2, t where this occurrence of t has 3 children whose tag is t', the (t, t', δ)-arc in whose label v_i^1 and v_i^2 occur must be one of the three (t, t', δ)-arcs whose label contains Q. All possible label elements have now been accounted for. The VDAG must have exactly three (t, t', δ)-arcs. The label of each of these three arcs must contain Q, plus some subset of tags corresponding to the members of V, with each v_i^1, v_i^2 pair occurring on exactly one of these three arcs. The occurrence of tags corresponding to each member of V in the label of exactly one of these three VDAG arcs represents an assignment of one of three colors to each member of V.

Now consider each edge e in E. Suppose $e = (v_i, v_j)$, where $i < j$. F contains a path r, v_i^1, v_j^2, t, where this occurrence of t has two children whose tags are t'. This requires that $\{v_i^1, v_i^2\}$ and $\{v_j^1, v_j^2\}$ occur in the labels of two distinct (t, t', δ)-arcs of the VDAG, i.e., v_i and v_j must be assigned different colors in G. Thus if F is VDAG-generable, then graph G is 3-colorable.

If G is 3-colorable, we can construct a labeling for (t, t', δ) as follows. For each set of vertices $v_{c_1}, \cdots, v_{c_k}$ of G which have the same color, let there be a VDAG arc



all arcs have stamp δ

FIG. 4.6

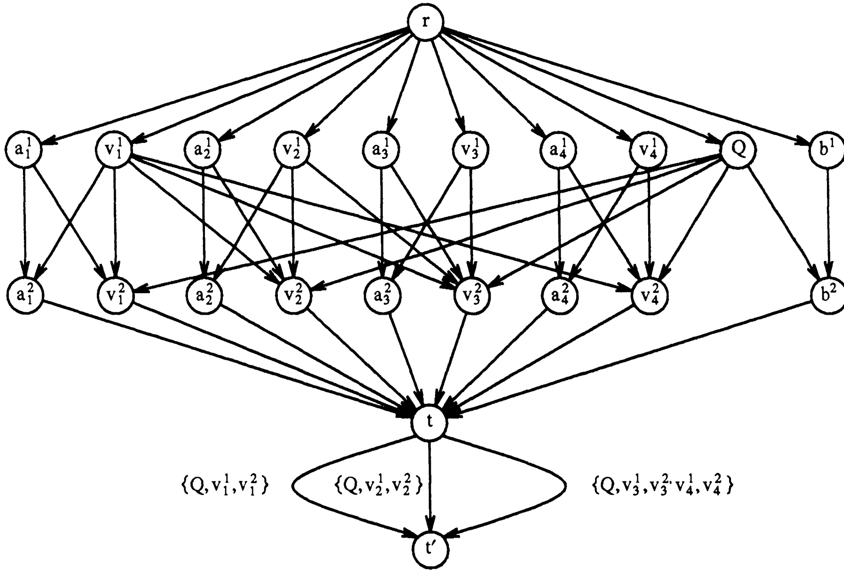


FIG. 4.7. *Arcs whose label is not shown have label* $\{r\}$. *All arcs have stamp* δ.

with label $\{Q, v_{c_1}^1, v_{c_1}^2, \cdots, v_{c_k}^1, v_{c_k}^2\}$. By the previous discussion, this is indeed a labeling for (t, t', δ). Hence F is VDAG-generable.

The above reduction can be easily extended to construct an F having any height $h \geqq 4$. □

In contrast to Theorem 4.3, the VDAG-generability of height 3 design trees can be determined in polynomial time, as shown in the following results.

DEFINITION. Given a design forest F, a relevant triple (t, t', δ) of F is *number-compatible* if for all vertices u and v of tag t, $pathtag(u) \subseteq pathtag(v)$ implies $num(u, t', \delta) \leqq num(v, t', \delta)$. F is *number-compatible* if every relevant triple of F is number-compatible.

It is easy to see that given a forest F, in polynomial time we can determine whether F is number-compatible. The following result shows that number-compatibility is a necessary condition for labelability.

LEMMA 4.4. *If a design forest is labelable, then it is number-compatible.*

Proof. Let (t, t', δ) be a relevant triple of a labelable design forest F. Since F is labelable, a number function $N_{t,t',\delta}$ exists, where for all vertices w of tag t in F,

$$\sum_{\gamma:\ \gamma \cap pathtag(w) \neq \varnothing} N_{t,t',\delta}(\gamma) = num(w, t', \delta).$$

Now consider two vertices of tag t in F, say u and v, such that $pathtag(u) \subseteq pathtag(v)$. Then for all $\gamma \in (2^{anctag(t)} - \varnothing)$, if $\gamma \cap pathtag(u) \neq \varnothing$ then $\gamma \cap pathtag(v) \neq \varnothing$. But this in turn implies that

$$\sum_{\gamma:\gamma \cap pathtag(u) \neq \varnothing} N_{t,t',\delta}(\gamma) \leqq \sum_{\gamma:\gamma \cap pathtag(v) \neq \varnothing} N_{t,t',\delta}(\gamma).$$

Hence $num(u, t', \delta) \leqq num(v, t', \delta)$, so (t, t', δ) is number-compatible. Since this is true for all relevant triples, F is number-compatible. □

Our next results show that a design tree of height at most 3 is VDAG-generable if and only if it is tag-acyclic and number-compatible.

LEMMA 4.5. *Let F be a tag-acyclic design tree of height at most* 3. *If F is number-compatible, then F is labelable.*

Proof. Let (t, t', δ) be a relevant triple of F, and let

$$minnum(t, t', \delta) = \min \{num(u, t', \delta) \mid u \text{ has tag } t\}.$$

If $maxnum(t, t', \delta) = minnum(t, t', \delta)$, then for all vertices u and v with tag t, $num(u, t', \delta) = num(v, t', \delta)$, and (t, t', δ) is labelable since we can specify a number function $N_{t,t',\delta}$ by assigning $N_{t,t',\delta}(\{t\}) = minnum(t, t', \delta)$, and for all nonempty $\gamma \subseteq anctag(t)$ where $\gamma \neq \{t\}$, $N_{t,t',\delta}(\gamma) = 0$.

So now suppose that $minnum(t, t', \delta) < maxnum(t, t', \delta)$, and we need to find a number function for (t, t', δ). Note that for any vertex u, with tag t, occurring at depth 0 or 1 in F, $num(u, t', \delta) = minnum(t, t', \delta)$. Also, since the height of F is at most 3, if there is a vertex u of depth 3 with tag t, this u has no children; so, in this case, $num(u, t', \delta) = 0 = minnum(t, t', \delta)$. Thus, every vertex u for which $num(u, t', \delta) > minnum(t, t', \delta)$ is of depth 2. Suppose q is the tag of a depth 1 vertex, having a (depth 2) child whose tag is t. Since F is number-compatible, all depth 2 vertices u having tag t and a parent whose tag is q have the same value for $num(u, t', \delta)$. Let this value be designated as $enum(q, t, t', \delta)$. For a tag q' that is not the tag of any depth 1 vertex having a child whose tag is t, let $enum(q', t, t', \delta)$ be undefined.

We now construct $N_{t,t',\delta}$ as follows. We assign $N_{t,t',\delta}(\{t\}) = minnum(t, t', \delta)$. For each q for which $enum(q, t, t', \delta)$ is defined, we assign $N_{t,t',\delta}(\{q\}) = enum(q, t, t', \delta) - minnum(t, t', \delta)$. For all other nonempty subsets γ of $anctag(t)$, we assign $N_{t,t',\delta}(\{\gamma\}) = 0$. By number-compatibility of F and our previous discussion, it is easily verified that $N_{t,t',\delta}$ is indeed a number function for (t, t', δ).

Since the above construction of a number function can be applied to all of the relevant triples of F, F must be labelable. □

THEOREM 4.6. *A design tree F of height at most* 3 *is* VDAG-*generable if and only if it is tag-acyclic and number-compatible.*

Proof. The proof is a direct result from Theorem 3.3 and Lemmas 4.4 and 4.5. □

THEOREM 4.7. *The* VDAG-*generability problem for trees of height at most* 3 *can be solved in polynomial time.*

Proof. Tag-acyclicity and number-compatibleness of design trees can each be tested in polynomial time. By Theorem 4.6, these tests are sufficient to determine whether a design tree of height 3 is VDAG-generable. □

5. Stamp uniqueness property and the effect of bounded stamp multiplicity. It is likely that in many design applications, for instance, applications in civil engineering design, mechanical design, or VLSI layout, the stamps on the arcs exiting from any given design tree vertex and entering vertices with the same tag would have to be distinct (e.g., it is not meaningful or useful in the design application for two instances of the same type of submodule to be placed in exactly the same position within a larger module). The class of design objects having this property is important, and covers the forests that would arise in many design systems. In this section we formalize this class of design forests, characterize the VDAGs that generate members of this class, and show that it is computationally easy to determine if a given forest in this class is VDAG-generable. This contrasts with the computational difficulty of determining if an arbitrary forest is VDAG-generable.

DEFINITION. Given a design forest F, let

$$stpmult(F) = \max_{u,t',\delta} \{num(u, t', \delta)\}.$$

Given a VDAG α, let

$$stpmult(\alpha) = \max\{m \mid \alpha \text{ contains vertices } u \text{ and } v \text{ such that there are } m \text{ arcs from } u \text{ to } v \text{ having the same stamp}\}.$$

Stpmult stands for *stamp multiplicity.*

We now show that determining whether a given forest is VDAG-generable is *NP*-complete, even for forests whose stamp multiplicity is 2.

THEOREM 5.1. *It is NP-complete to determine whether a given forest F, where* $stpmult(F) = 2$, *is* VDAG-*generable.*

Proof. The argument for membership in *NP* is the same as in the proof of Theorem 4.3. The *NP*-hardness proof is by a reduction from Not-All-Equal 3SAT, which is known to be *NP*-complete [4]. A Not-All-Equal 3SAT instance consists of a set of variables X and a set of clauses C, each of which contains three literals. The computational problem is whether there is a truth assignment to the variables such that each clause has at least one literal *true* and at least one literal *false.*

Consider a given Not-All-Equal 3SAT instance with $X = \{x_1, x_2, \cdots, x_n\}$ and $C = \{c_1, c_2, \cdots, c_m\}$, where for each j, $1 \leqq j \leqq m$, c_j contains literals l_j^1, l_j^2, and l_j^3, and the index of the variable corresponding to l_j^1 is less than the index of the variable corresponding to l_j^2, which in turn is less than the index of the variable corresponding to l_j^3. Construct the forest F consisting of the single tree whose form is shown in Fig. 5.1. For example, given the Not-All-Equal 3SAT instance shown in Fig. 5.2, the constructed forest F is shown in Fig. 5.3, and a VDAG generating F is shown in Fig. 5.4. Note that $stpmult(F) = 2$, F is tag-acyclic, and the size of F is linear in m and n. Also note that there is a number function for each relevant triple, with the possible exception of (A, B, δ).

Suppose the forest is VDAG-generable, and consider relevant triple (A, B, δ). Consider the occurrences of A in paths S, W, A; S, Y, A; S, Z, A and for each j, $1 \leqq j \leqq m$, S, c_j, A. Since each such A has no child with tag B, it follows that S, W, Y, Z, c_j, and A cannot appear in the label of any VDAG arcs from A to B. Consider paths

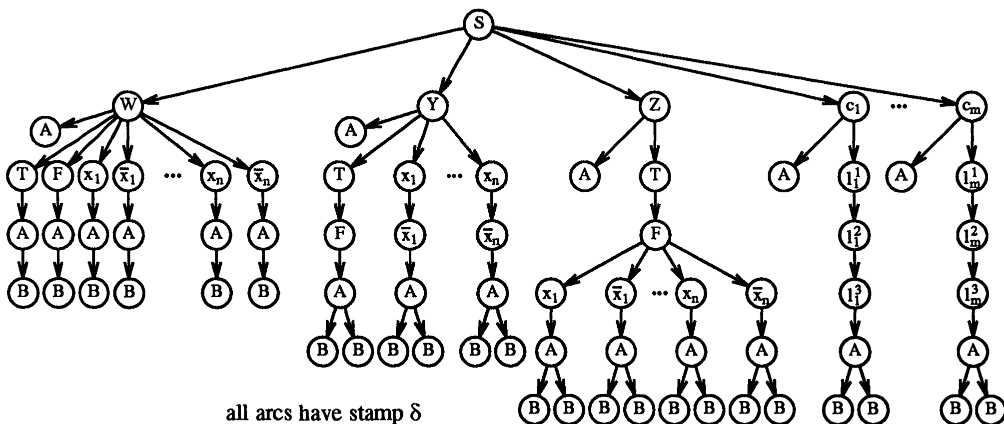


FIG. 5.1

$$X = \{x_1, x_2, x_3\}$$

$$C = \{(x_1, \bar{x}_2, x_3), (x_1, x_2, \bar{x}_3)\}$$

FIG. 5.2

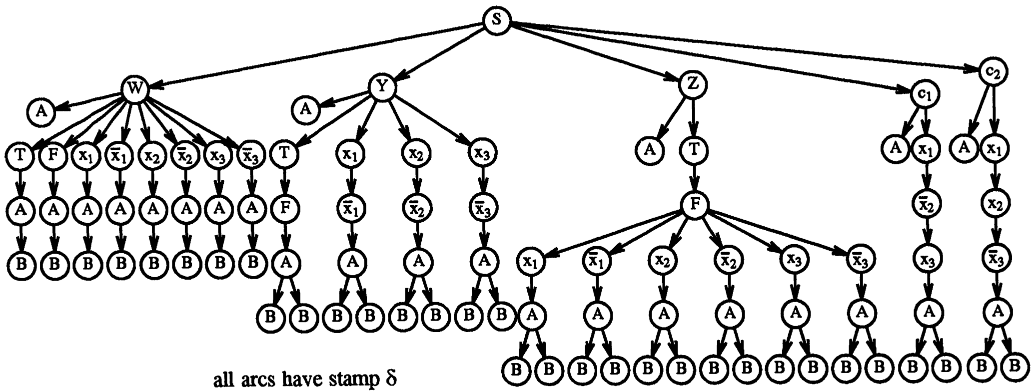


FIG. 5.3

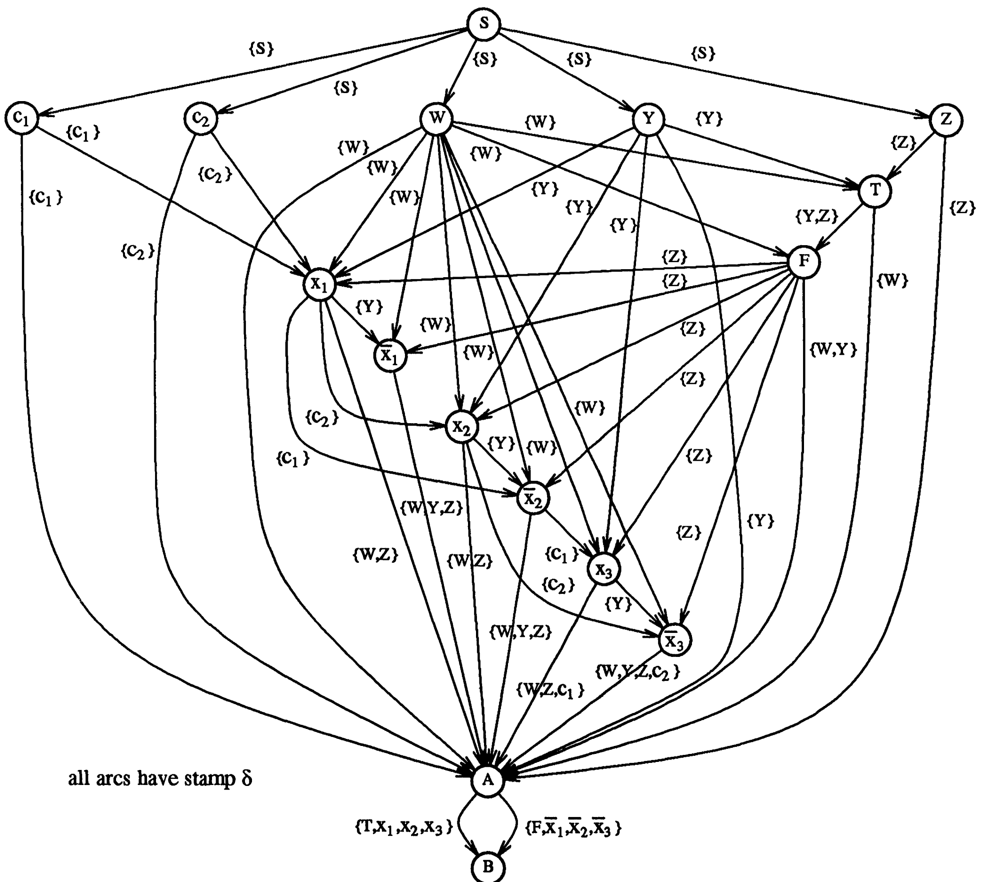


FIG. 5.4

S, W, T, A; S, W, F, A and for each $i, 1 \leqq i \leqq n$, S, W, x_i, A and $S, W, \bar{x}_i, A$. Since each occurrence of A in each of these paths has exactly one child with tag B, it follows that $T, F, x_i, \bar{x}_i$ each appears in the label of exactly one VDAG arc from A to B. Consider paths S, Y, T, F, A and $S, Y, x_i, \bar{x}_i, A$, where each occurrence of A has two children with tag B. It follows that T and F have to be in two different labels; and similarly, x_i and $\bar{x}_i$ have to be in two different labels. Because each occurrence of A on paths S, Z, T, F, x_i, A and $S, Z, T, F, \bar{x}_i, A$ has two children with tag B, x_i and $\bar{x}_i$ must be in the labels containing T or F. It follows that there must be exactly two VDAG arcs with stamp δ from A to B. Now consider, for each $j, 1 \leqq j \leqq m$, the occurrence of A in the path $S, c_j, l_j^1, l_j^2, l_j^3, A$. This A has two children with tag B, hence at least one of the three literals must appear in the label containing T and at least one of the three literals must appear in the label containing F. We may interpret the literals in the label containing T as being assigned *true* and the literals in the label containing F as being assigned *false* in the truth assignment for the Not-All-Equal 3SAT instance. Thus the appearance of literals in the two arcs from A to B in a VDAG generating F represent a not-all-equal satisfying truth assignment.

By the above discussion, the given Not-All-Equal 3SAT has a satisfying truth assignment if and only if there is a labeling on VDAG arcs from A to B, that is, if and only if F is VDAG-generable. □

DEFINITION. Let *UnqStamp* be the set of design forests such that no two arcs exiting from the same vertex and entering vertices with the same tag have the same stamp, i.e., forests whose stamp multiplicity is 1. Let $VD_{UnqStamp}$ be the set of VDAGs such that no two arcs with the same endpoints have the same stamp, i.e., VDAGs whose stamp multiplicity is 1.

Later we will show that there is a close correspondence between *UnqStamp* and $VD_{UnqStamp}$.

We now consider VDAG-generability problem for forests in *UnqStamp*. First, note that from Theorem 3.3, a given member of *UnqStamp* is VDAG-generable if and only if it is well structured. To be well structured, the design forest must be both tag-acyclic and labelable. Since even for arbitrary forests, tag-acyclicity is testable in linear time, the main computational issue is labelability. The forests in Figs. 3.1(a) and (b) are examples of forests in *UnqStamp* that are tag-acyclic, but not labelable, and therefore not VDAG-generable. In both these examples, a number function for relevant triple (B, C, δ) does not exist because the forest contains vertices u and v with tag B such that $num(u, C, \delta) = 1$, $num(v, C, \delta) = 0$, and $pathtag(u)$ is contained in $pathtag(v)$. This phenomenon can serve as the basis for a polynomial time test for labelability, as follows.

LEMMA 5.2. *The problem of determining whether a member of UnqStamp is* VDAG-*generable can be solved in polynomial time.*

Proof. Consider a given forest F in *UnqStamp*. Using the algorithm COLLAPSE from Proposition 4.1, F can be tested for tag-acyclicity in linear time. If F is not tag-acyclic, then it is not VDAG-generable. Thus, assume F is tag-acyclic, and consider its labelability.

Note that since F is in *UnqStamp*, for any vertex u, tag t', and stamp δ, $num(u, t', \delta)$ is at most 1. Given a relevant triple (t, t', δ), let the set of all paths from a root to a vertex u with tag t, where $num(u, t', \delta) = 0$ be $P_0(t, t', \delta)$, and the set of all paths from a root to a vertex u with tag t, where $num(u, t', \delta) = 1$ be $P_1(t, t', \delta)$. Let $\Gamma_0(t, t', \delta)$ be the set of all tags occurring in the paths in $P_0(t, t', \delta)$. Similarly, let $\Gamma_1(t, t', \delta)$ be the set of all tags occurring in the paths in $P_1(t, t', \delta)$. For each p in $P_1(t, t', \delta)$, we check if the set of tags of vertices on path p is contained in $\Gamma_0(t, t', \delta)$. If one such p exists,

$N_{t,t',\delta}$ cannot exist, and F is not labelable. Otherwise, let $N_{t,t',\delta}(\Gamma_1(t, t', \delta) - \Gamma_0(t, t', \delta)) = 1$, and for all $\gamma \neq \Gamma_1(t, t', \delta) - \Gamma_0(t, t', \delta)$, let $N_{t,t',\delta}(\gamma) = 0$. It is easy to see that $N_{t,t',\delta}$ satisfies the requirements for a number function. If for each relevant triple (t, t', δ), $N_{t,t',\delta}$ is found, F is labelable. Otherwise, F is not labelable. Note that the above test can be done in polynomial time. □

If a forest F in *UnqStamp* is well structured, then Algorithm VDAGCONS, shown in Fig. 5.5, constructs a VDAG that generates F.

ALGORITHM VDAGCONS
Input: a forest F in *UniStamp*
Output: a VDAG α generating F, if F is well structured
1. if F is not tag-acyclic, **exit**;
2. apply the algorithm used in the proof of Lemma 5.2 to F;
   if F is not well structured, **exit**;
3. let α have a vertex for each tag occurring in F;
   for relevant triple (t, t', δ) **do**
      let α have an arc a with stamp δ going from the vertex with tag t to the vertex with tag t', and label arc a with $(\Gamma_1(t, t', \delta) - \Gamma_0(t, t', \delta))$ as described in the proof of Lemma 5.2.

FIG. 5.5

THEOREM 5.3. *Given a member of UnqStamp, say F, we can in polynomial time,* (1) *determine whether F is* VDAG-*generable*; *and* (2) *if F is indeed* VDAG-*generable, construct a* VDAG *that generates F.*

Proof. Algorithm VDAGCONS terminates in polynomial time. If it terminates by *exit*, the given forest is not VDAG-generable, as implied by the proof of Lemma 5.2; otherwise, it is VDAG-generable. If Algorithm VDAGCONS constructs a VDAG, then from the proof of Lemma 5.2, this VDAG generates F. □

We next note that the algorithm in the proof of Lemma 5.2 is applicable not only to forests in *UnqStamp*, but to any relevant triples for which the value of *maxnum* is 1, even if other relevant triples have a larger value of *maxnum*.

DEFINITION. A relevant triple (t, t', δ) for a design forest F has the *unique stamp property* if $maxnum(t, t', \delta) = 1$.

THEOREM 5.4. *If a relevant triple* (t, t', δ) *for a forest F has the unique stamp property, then we can in polynomial time* (1) *determine if* (t, t', δ) *is labelable*; *and* (2) *if it is, construct a number function for* (t, t', δ).

Proof. Apply the algorithm from the proof of Lemma 5.2. □

We now show that the set of forests generated by members of $VD_{UnqStamp}$ is identical to the set of VDAG-generable members of *UnqStamp*.

THEOREM 5.5. *The set of forests generated by members of* $VD_{UnqStamp}$ *is precisely the set of forests in UnqStamp which are* VDAG-*generable.*

Proof. Consider a forest F which is a well-structured member of *UnqStamp*. Then, $\alpha = \text{VDAGCONS}(F)$ is a VDAG generating F. Since for each relevant triple (t, t', δ), the constructed VDAG α has exactly one (t, t', δ)-arc, α is a member of $VD_{UnqStamp}$.

Now consider a member α of $VD_{UnqStamp}$. No two arcs of α with the same endpoints have the same stamp; therefore, if a vertex of forest F_α has more than one child with the same tag, the stamps on the arcs from the vertex to these children must have distinct stamps. Hence F_α is a member of *UnqStamp*. □

6. Construction of number functions. Theorems 4.3 and 5.1 imply that finding a labeling for a forest is *NP*-hard. If $P \neq NP$, this task cannot be done in polynomial

time. However, finding a labeling can be divided into a set of independent subtasks, namely finding a number function for each relevant triple (t, t', δ). In searching for a number function for a given relevant triple, an obvious approach is to enumerate potential number functions in a canonical order, and check whether the labelable condition is satisfied. The search space can be reduced because if a given forest is labelable, its number functions are "sparse," in a sense formalized by the following result. The next result shows that if there is a number function, then there is one in which $N_{t,t',\delta}(\gamma)$ is nonzero for only a small number of sets γ.

THEOREM 6.1. *If a relevant triple (t, t', δ) for a forest F is labelable, then it has a number function $N_{t,t',\delta}$ for which*

$$\sum_{\gamma \in (2^{anctag(t)} - \varnothing)} (|\gamma| \cdot N_{t,t',\delta}(\gamma)) \leqq totnum(t, t', \delta).$$

Proof. Let α be a valid VDAG generating F. For each relevant triple (t, t', δ), consider the (t, t', δ)-arcs in α. From the proof of Theorem 4.1, each (t, t', δ)-arc in F can be associated with one of these arcs in α; and each of these arcs in α is associated with at least one of these arcs in F. This association can be strengthened by associating each F arc with a single element of the label of its associated α arc. In particular, since an F arc exiting a vertex u is associated with an α arc having label γ such that $\gamma \cap pathtag(u)$ is nonnull, let the F arc be associated with some arbitrary member of $\gamma \cap pathtag(u)$. Note that because α is valid, each arc label in α will have at least one F arc associated with at least one of the elements of the label.

Now, modify the arc labels in α by deleting every label element that has no F arc associated with it. Let β be the resulting VDAG. Then β is a valid VDAG that generates F. Let $N_{t,t',\delta}$ be the number function for (t, t', δ) that is embodied in β. Each arc of F that contributes to $totnum(t, t', \delta)$ is associated with some label element in β, and each label element is associated with at least one such arc. A label γ in β is, therefore, associated with at least $|\gamma|$ arcs of F. Since there are only $totnum(t, t', \delta)$ arcs available for this association, it is the case that

$$\sum_{\gamma \in (2^{anctag(t)} - \varnothing)} (|\gamma| \cdot N_{t,t',\delta}(\gamma)) \leqq totnum(t, t', \delta). \qquad \square$$

COROLLARY 6.2. *If a given design forest F is* VDAG-*generable, then there exists a* VDAG *α generating F whose size is linearly bounded by the size of F.*

Proof. First, note that the number of vertices in a valid VDAG generating F cannot exceed the number of distinct tags on vertices in F. Theorem 6.1 implies that the total number of occurrences of label elements in α need not exceed the number of vertices in F. Since arc labels are nonempty, the total number of arcs cannot exceed the total number of label elements, and thus cannot exceed the total number of vertices in F. $\square$

Another observation that can sometimes reduce the search space for a number function is that the members of $anctag(t)$ for tag t can be partitioned on the basis of occurrences in paths, and only one representative from each block of the partition need be considered. This concept can be formalized as follows.

DEFINITION. Given a forest F and a tag t, two tags p, q in $anctag(t)$ are *equivalent* with respect to t if for every vertex u such that $t(u) = t$, either p and q are both in $pathtag(u)$ or neither is. Let $neqtag(t)$ be a set that contains exactly one member from each tag equivalence class with respect to t.

For each relevant triple (t, t', δ), the search space for a number function $N_{t,t',\delta}$ can be reduced to nonnull subsets of $neqtag(t)$, instead of $anctag(t)$, as shown by the following result.

DEFINITION. For each γ that is a subset of $anctag(t)$, let $neq(\gamma)$ be the subset of $neqtag(t)$ defined as follows: a member w of $neqtag(t)$ is in $neq(\gamma)$ if and only if γ contains at least one member of the equivalence class represented by w.

LEMMA 6.3. *If a relevant triple (t, t', δ) for a forest F is labelable, then it has a number function for which each argument having a nonzero value is a nonnull subset of $neqtag(t)$.*

Proof. Suppose that there is a number function $N_{t,t',\delta}$ for (t, t', δ). On the basis of this number function, define a function $M_{t,t',\delta}$ from $(2^{anctag(t)} - \varnothing)$ to $\mathbf{N}$ as follows. For each nonnull subset ξ of $neqtag(t)$, let

$$M_{t,t',\delta}(\xi) = \sum_{\gamma: neq(\gamma)=\xi} N_{t,t',\delta}(\gamma).$$

For each set ξ that contains a member of $(anctag(t) - neqtag(t))$, let $M_{t,t',\delta}(\xi) = 0$.

Note that from the definition of the equivalence relation used to construct $neqtag(t)$, for each vertex u with tag t in F, and each subset γ of $anctag(t)$, $\gamma \cap pathtag(u)$ is nonnull if and only if $neq(\gamma) \cap pathtag(u)$ is nonnull; therefore, since $N_{t,t',\delta}$ is a number function for (t, t', δ), so is $M_{t,t',\delta}$. Furthermore, $M_{t,t',\delta}$ only has a nonzero value for subsets of $neqtag(t)$. □

The search space for a number function can be reduced even further, as follows.

DEFINITION. Given a forest F and a relevant triple (t, t', δ), let $\Gamma_0(t, t', \delta)$ be the set of tags that occur in $pathtag(u)$ for some vertex u of the forest having tag t and for which $num(u, t', \delta)$ equals zero. Let $candtag(t, t', \delta)$ be $(neqtag(t) - \Gamma_0(t, t', \delta))$.

Note that a member of $\Gamma_0(t, t', \delta)$ cannot be a member of any set γ for which a number function for (t, t', δ) has a nonzero value. Thus we can confine the search for a number function for (t, t', δ) to nonnull subsets of $candtag(t, t', \delta)$. This can be formalized as follows.

THEOREM 6.4. *If a relevant triple (t, t', δ) for a forest F is labelable, then it has a number function for which each argument having nonzero value is a nonnull subset of $candtag(t, t', \delta)$.*

The search space can be reduced still further by observing that two vertices in a forest having the same tag t and whose ancestors include the same subset of $candtag(t, t', \delta)$ are equivalent with respect to finding a labeling. If forest F contains two vertices u and v with tag t such that $pathtag(u) \cap candtag(t, t', \delta) = pathtag(v) \cap candtag(t, t', \delta)$, then for (t, t', δ) to be labelable it must be the case that $num(u, t', \delta) = num(v, t', \delta)$. Suppose that this condition is not violated for relevant triple (t, t', δ). Let $\psi(t, t', \delta)$ be the collection of subsets of $candtag(t, t', \delta)$ such that γ is in $\psi(t, t', \delta)$ if and only if F contains a vertex u with tag t such that $\gamma = pathtag(u) \cap candtag(t, t', \delta)$. Also, let $numc(\gamma, t, t', \delta)$ be the value of $num(u, t', \delta)$ for some vertex u with tag t for which $\gamma = pathtag(u) \cap candtag(t, t', \delta)$. Let

$$totnumc(t, t', \delta) = \sum_{\gamma \in \psi(t, t', \delta)} numc(\gamma, t, t', \delta).$$

Then, from the reasoning used in the proof of Theorem 6.1, we have the following result.

THEOREM 6.5. *If a relevant triple (t, t', δ) for a forest F is labelable, then it has a number function $N_{t,t',\delta}$ that only has nonzero values for subsets of $candtag(t, t', \delta)$, and for which*

$$\sum_{\gamma \in (2^{candtag(t,t',\delta)} - \varnothing)} (|\gamma| \cdot N_{t,t',\delta}(\gamma)) \leqq totnumc(t, t', \delta).$$

7. Handling non–VDAG-generable forests. We pointed out earlier that a given forest that is not well structured can be made well structured by modifying the tags of its vertices. This concept can be formalized as follows.

DEFINITION. A *homomorphism* h from design forest F *onto* design forest G is a mapping from the tags occurring in F onto the tags occurring in G, such that applying mapping h to F produces G. A VDAG α *embodies* design forest G if there exists a homomorphism from F_α onto G.

Note that a homomorphism can map several tags of F onto the same tag of G. From a different perspective, G is the same as F, except that each tag of G has been split into one or more tags in F and for each vertex of G the corresponding vertex in F has as its tag some member of the split tag set.

An observation is that if a given forest F is modified to have a distinct tag on each of its vertices, then there is a homomorphism from the modified forest F^* to F. Furthermore, F^* is well structured, and each valid VDAG generating F^* is simply F^* with appropriate labels on its arcs. One such labeling scheme is for each arc exiting a vertex with tag t to be assigned label $\{t\}$. We thus see that every forest is embodied by some VDAG. Furthermore, it is embodied by a VDAG whose expansion is unconditional, where VDAGs with unconditional expansion, corresponding to the conventional dag representation of hierarchically specified design objects, can be formalized as follows.

DEFINITION. An *UncVDAG* is a VDAG where the label of each arc contains only the tag of the vertex exited by the arc.

"UncVDAG" stands for *Uncontrolled-expansion VDAG.* The UncVDAGs are a subclass of VDAGs, corresponding to conventional dags with uncontrolled expansion. The only difference between a UncVDAG and a conventional dag is that an UncVDAG has the extra overhead of storing a one-element label for each arc.

From the preceding discussion, we have the following observation.

PROPOSITION 7.1. *Every design forest is embodied by some* UncVDAG.

A natural computational problem is minimizing the amount of splitting needed to obtain a VDAG embodying a given forest. This problem can be posed as a decision problem, as follows.

MINIMUM TAG SPLIT (MTS).

Instance: A design forest F with m distinct tags $t_1, t_2, \cdots, t_m$, positive integers sp_i, $1 \leqq i \leqq m$.

Question: Is there a VDAG generating design forest F^*, where F^* is obtained from F via replacing tag t_i by at most sp_i modified versions of t_i, $1 \leqq i \leqq m$?

The following result shows that this problem is NP-complete.

THEOREM 7.2. MTS *is NP-complete.*

Proof. The membership of MTS in NP is obvious. The NP-hardness proof is by a simple reduction from the problem: "Given a forest F, determine whether there is a VDAG generating F," which has been shown to be NP-complete by Theorem 4.3. For a given forest F with m distinct tags, we construct an MTS instance by assigning $sp_i = 1$ for all i, $1 \leqq i \leqq m$. It is straightforward to verify that there is a positive answer for the constructed MTS instance if and only if there exists a VDAG generating F. □

8. Relative conciseness of VDAG model. As mentioned in § 1, one of the purposes of the VDAG model is to succinctly represent design versions and alternatives. It is well known that conventional dags can be exponentially more succinct than trees. For instance, consider a UncVDAG β_n with n vertices, numbered $v_1, v_2, \cdots, v_n$, such that v_i has two arcs going to v_{i+1}, for $1 \leqq i < n$. Then F_{β_n} is a tree with $2^n - 1$ vertices; so F_{β_n} is exponentially larger than β_n.

We now show that VDAGs, utilizing controlled expansion, can be exponentially more succinct than UncVDAGs, where expansion is uncontrolled.

Given a positive integer n, let VDAG α_n be that shown in Fig. 8.1. All stamps in α_n are δ. The label on the arc from B_n to C_i is $\{A_i\}$, $1 \leqq i \leqq n$, and all the remaining arcs are labeled by $\{R\}$. It is easily verified that there are $3n+1$ vertices, $4n$ arcs and $4n$ label elements in α_n.

Now consider F_{α_n}. It is easily seen that there are 2^n generating paths in α_n from R to B_n, where for each i, $1 \leqq i \leqq n$, each such path either goes through or bypasses A_i. In particular, each of these 2^n generating paths to B_n goes through a distinct subset of the A_i vertices. The instance of B_n in F_{α_n} corresponding to a generating path in α_n, say path p, has an instance of C_i as a child if and only if p goes through A_i. Hence F_{α_n} contains 2^n distinct subtrees rooted by instances of B_n. The leaves of each of these subtrees correspond to a distinct member of the power set of $\{C_1, C_2, \cdots, C_n\}$. More generally, for each i, $1 \leqq i \leqq n$, there are 2^i distinct subtrees rooted by instances of B_i and 2^{i-1} distinct subtrees rooted by instances of A_i. For example, Fig. 8.2 shows F_{α_2}.

Let D be a UncVDAG embodying F_{α_n}. Since for each i, $1 \leqq i \leqq n$, each of the 2^i vertices with tag B_i in F_{α_n} is the root of a distinct subtree, D must split tag B_i into 2^i tags, i.e., D must have a separate vertex for each of these 2^i vertices of F_{α_n}. Similarly, D must have a separate vertex for each of the 2^{i-1} vertices with tag A_i in F_{α_n}. Thus D must have at least

$$\sum_{i=0}^{n-1} 2^i + \sum_{i=1}^{n} 2^i + n + 1 = 3 \cdot 2^n + n - 2$$

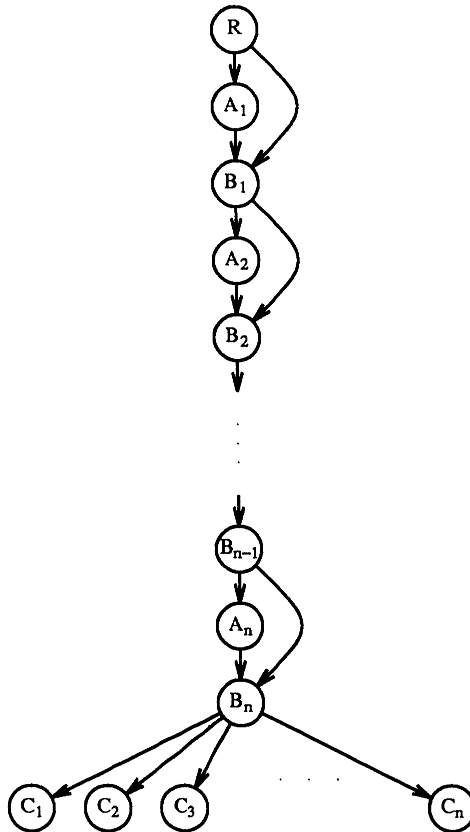


FIG. 8.1. *All arcs have stamp* δ.

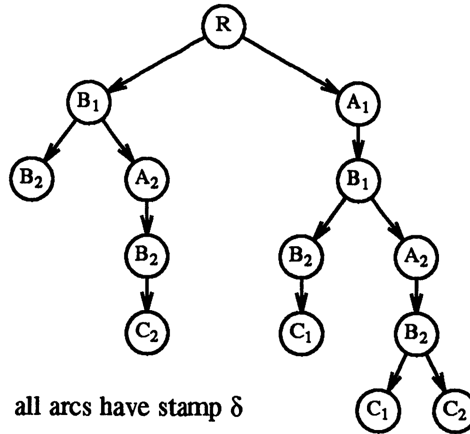


FIG. 8.2

vertices. Since there must be

$$\sum_{i=0}^{n} i \cdot \binom{n}{i} = n \cdot 2^{n-1}$$

arcs exiting from instances of B_n and entering instances of C, and each instance of A and each instance of B must have exactly 1 entering arc, D must have

$$\sum_{i=0}^{n} i \cdot \binom{n}{i} + \sum_{i=0}^{n-1} 2^i + \sum_{i=1}^{n} 2^i = (n+6) \cdot 2^{n-1} - 3$$

arcs.

We have just shown the following result.

THEOREM 8.1. *For each positive integer n, there exists a* VDAG α_n *with* $3n+1$ *vertices,* $4n$ *arcs, and* $4n$ *label elements, such that every* UncVDAG *embodying* F_{α_n} *has at least* $(n+6) \cdot 2^{n-1} - 3$ *arcs and at least* $3 \cdot 2^n + n - 2$ *vertices.*

Therefore, there exists design objects whose VDAG representation is exponentially more succinct than their representation using a conventional dag with uncontrolled expansion. On the other hand, if design versions and alternatives are not involved, so that controlled expansion is not needed, the only extra overhead of VDAG with respect to conventional dags is a one-element label per arc. Since the size of a tag identifier is usually smaller than the size of a stamp, this overhead will usually be small. This overhead could be reduced even further by adopting a default convention that an arc is to be included unconditionally, unless the label says otherwise.

9. Automatic simplification. In this section we consider the problem of, given a well-structured forest F, finding a VDAG that generates F and has as small a size as possible.

DEFINITION. Two VDAGs are *equivalent* if the forests they produce are identical. VDAG α is *smaller* than VDAG β if α has fewer arcs than β, or if α and β have the same number of arcs and α has a smaller total label size than β. A VDAG α is *minimum-sized* if there is no equivalent VDAG β such that β is smaller than α.

Since the size of a VDAG is finite, for each VDAG there exists some equivalent minimum-sized VDAG.

THEOREM 9.1. *For each* $h \geqq 5$, *given a well-structured design tree F in UnqStamp with height h, it is NP-hard to find a minimum-sized* VDAG *generating F.*

Proof. We show NP-hardness by a reduction from the Vertex Cover Problem [4]. The Vertex Cover decision problem consists of determining for a given undirected graph G and integer k, whether there is a set of at most k vertices such that each edge is incident on at least one vertex in the set. Given a Vertex Cover instance involving graph G with the set of vertices $V=\{v_1, v_2, \cdots, v_n\}$ and the set of edges $E=\{e_1, e_2, \cdots, e_m\}$, we construct a forest F as follows. F consists of one tree where the root has tag S. For each i, $1 \leqq i \leqq m$, root S has a child with tag e_i. Suppose the two endpoints of e_i are v_{i_1} and v_{i_2}, where $i_1 < i_2$. Then e_i has two children: one with tag A and the other with tag v_{i_1}. This vertex with tag v_{i_1} has a child with tag v_{i_2}, which in turn has a child with tag A, which has a child with tag B. Stamp δ is assigned to all arcs in F. F is in *UnqStamp* since no two children of a vertex have the same tag. Note that the height of F is 5.

For example, given the Vertex Cover instance shown in Fig. 4.5, the constructed forest is shown in Fig. 9.1, and a VDAG generating that forest is shown in Fig. 9.2.

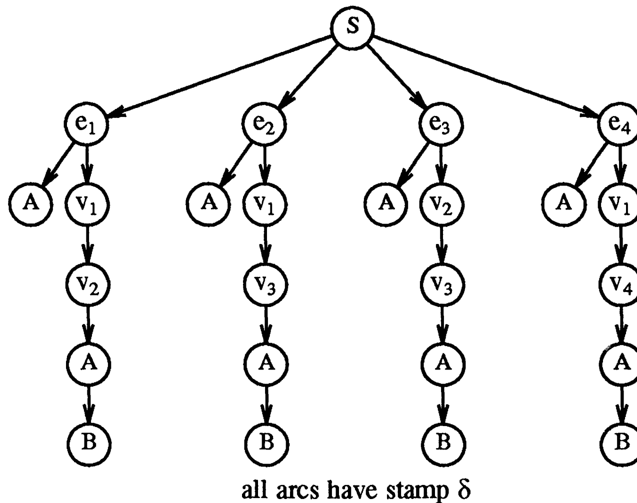


FIG. 9.1

To see that the constructed F is well structured in general, first note that F is tag-acyclic. Next, consider labelability. For relevant triples of the form (S, e_i, δ), the label can be $\{S\}$. For relevant triples whose first component is e_i, the label can be $\{e_i\}$. For relevant triples of the form (v_i, v_j, δ), $i<j$, the label can be $\{e_k\}$, where e_k is the edge of G which is incident on both v_i and v_j. For relevant triples of the form (v_j, A, δ), the label can be the set of e's that have v_i and v_j as endpoints for some $i<j$. For relevant triple (A, B, δ), the label can be V. Thus F is well structured, and hence VDAG-generable.

Theorem 5.5 asserts that there is a VDAG in $VD_{UnqStamp}$ that generates F. Hence in each minimum-sized VDAG generating F, only one (A, B, δ)-arc is needed. Let α be a minimum-sized VDAG in $VD_{UnqStamp}$, and let the label on the (A, B, δ)-arc in α be l. For each i, $1 \leqq i \leqq m$, because of the path S, e_i, A in F, where this occurrence of A has no children, S, A, and e_i do not occur in l. Hence only v_i's can appear in l, $1 \leqq i \leqq n$. Note that a minimum-sized VDAG generating the forest in Fig. 9.1 may have $l=\{v_1, v_3\}$, and $\{v_1, v_3\}$ is a minimum cover for the given vertex cover instance in Fig. 4.5. In general, for each e_i, the generating path in α from S to e_i to v_{i_1} to v_{i_2} to A must involve at least one member of l. Therefore, l contains at least one of $\{v_i, v_{i_2}\}$.

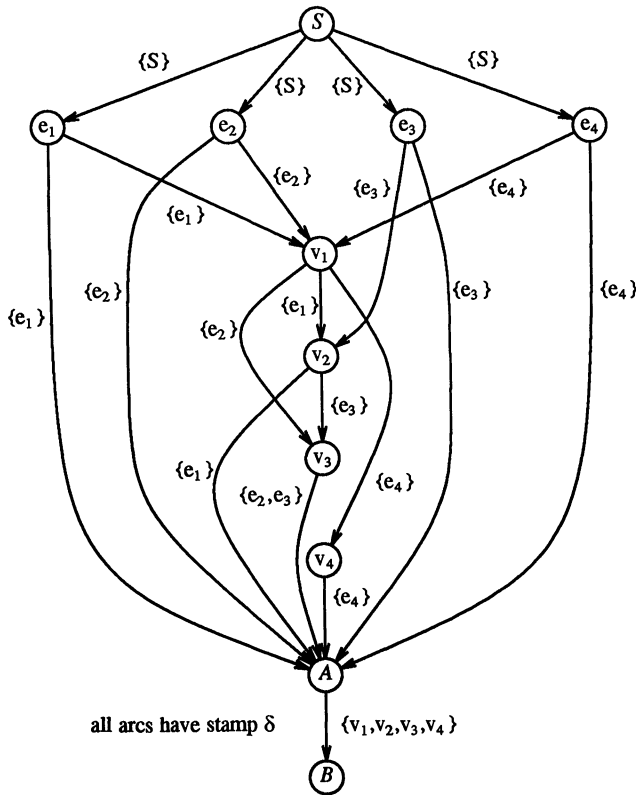


FIG. 9.2

Hence, set l covers all elements of E and is indeed a vertex cover; therefore, there exists a k-element label for arc (A, B) if and only if graph G has a vertex cover of size k.

The above reduction can be easily extended to construct an F having any height $h \geqq 5$. □

From Theorem 9.1, we obtain the following results.

COROLLARY 9.2. *For each* $h \geqq 4$, *given a well-structured design forest in UnqStamp of height h, it is NP-hard to find a minimum-sized* VDAG *generating that forest.*

Proof. The proof is similar to that in Theorem 9.1, with the root and its associated arcs removed from the constructed tree. □

In contrast to Theorem 9.1, minimum-sized VDAGs can be efficiently found for VDAG-generable trees of height 4 or less.

THEOREM 9.3. *For each* VDAG-*generable tree F in UnqStamp of height* 4 *or less, a minimum-sized* VDAG *generating F can be found in polynomial time.*

Proof. Let (t, t', δ) be a relevant triple. Then by Theorem 5.5, a minimum-sized VDAG generating F has exactly one (t, t', δ)-arc. A minimum size label for this arc must be found.

If $\Gamma_0(t, t', \delta)$ is null, then $\{t\}$ can serve as a minimum size label for this arc. Thus assume that $\Gamma_0(t, t', \delta)$ is nonnull. Then both t and the tag of the root vertex are in $\Gamma_0(t, t', \delta)$, and so neither can be elements of the label.

Let *relvert* be the set of vertices u of F such that the tag of u is t and $num(u, t', \delta) = 1$. Note that each member of *relvert* has depth 2 or 3 in F.

For vertex u of *relvert*, let

$$ptag(u) = pathtag(u) - \Gamma_0(t, t', \delta).$$

Note that $|ptag(u)|$ must be either 1 or 2. Let *onecand* be the set of members u of *relvert* such that $|ptag(u)| = 1$. Let

$$mustintag = \bigcup_{u \text{ in } onecand} ptag(u).$$

Note each member w of *mustintag* is a tag that is the only ancestor of some member u of *relvert*, such that $\{w\} = pathtag(u) - \Gamma_0(t, t', \delta)$, so w must be an element of the label being constructed.

For a vertex u of *relvert*, let

$$ctag(u) = ptag(u) - mustintag.$$

Note that if $|ctag(u)| = 1$, then one of the ancestors of u is a member of *mustintag*, and the presence of this tag in the label will account for vertex u. If $|ctag(u)| = 0$, then u is a member of *onetag*, so an ancestor of u is a member of *mustintag*.

Let *twocand* be the set of members u of *relvert* such that $|ctag(u)| = 2$. The members of *twocand* present a design choice in constructing the label. Each member u of *twocand* must be accounted for by having at least one member of $ctag(u)$ included in the label, but it is not necessary for both members of $ctag(u)$ to be included. This gives rise to a Vertex Cover Problem as follows.

We construct a Vertex Cover instance $G(V, E)$ as follows. Let

$$V = \bigcup_{u \text{ in } twocand} ctag(u).$$

For each u in *twocand*, E contains an undirected edge between the two members of $ctag(u)$. A minimum size label consists of the union of *mustintag* and a minimum size vertex cover of graph G.

Now, we show that graph G is bipartite. First note that for each u in *twocand*, the two vertices of F that contribute to $ctag(u)$ occur at depth 1 or 2, respectively. Let A be a member of V. Suppose that there is a vertex at depth 1 with tag A that is an ancestor of a member of *twocand*. Then, there are tags S and B such that S, A, B, t, t' are the tags along a path in F. Suppose that there is also a vertex at depth 2 with tag A that is an ancestor of a member of *twocand*. Then there is a tag C such that S, C, A, t, t', are the tags along a path in F. Let α be a VDAG generating F; since F is VDAG-generable, such a VDAG exists. Because of the two forest paths described above, VDAG α must have an arc from S to C labeled $\{S\}$, an arc from C to A whose label contains S or C, an arc from A to B whose label contains S or A, an arc from B to t whose label contains S, A or B, and an arc from t to t' whose label contains A or B. But then α contains a generating path with tags S, C, A, B, t, t'. This generating path would correspond to a depth 5 vertex in F, contradicting the assumption that F has height 4. Thus either all the vertices with tag A that are ancestors of members of *twocand* have depth 1, or they all have depth 2. Consequently, the constructed graph G is bipartite, with each edge of G connecting a depth 1 tag and a depth 2 tag.

Given a bipartite graph, a minimum size vertex cover can be found in polynomial time [4], [5]. Therefore, a minimum size label for (t, t', δ) can be found in polynomial time. □

An immediate consequence is the following.

COROLLARY 9.4. *For* VDAG-*generable design forests in UnqStamp of height at most* 3, *a minimum-sized* VDAG *generating a given forest can be found in polynomial time.*

We now consider the issue of whether the value of $maxnum(t, t', \delta)$ for a relevant triple in a forest can be used to limit the number of VDAG (t, t', δ)-arcs. The next result shows that the number of such arcs in the VDAG may have to be arbitrarily larger than $maxnum(t, t', \delta)$, that is, in general, the stamp multiplicity of a VDAG cannot be bounded by the stamp multiplicity of the forest it generates.

THEOREM 9.5. *For each integer* $k \geqq 2$, *there exists a* VDAG-*generable forest* F, *such that* $stpmult(F) = 2$ *and for each* VDAG α *generating* F, $stpmult(\alpha) = k$.

Proof. For a given k, consider the complete undirected graph of k vertices, and consider the forest F_k produced from this graph by the construction used in the proof of Theorem 4.3, except with the subtree rooted by Q removed. It is easily seen that $stpmult(F_k) = 2$ and that F_k is VDAG-generable. Suppose that VDAG α generates F_k. By an argument similar to that used in the proof of Theorem 4.3, it can be seen that the number of (t, t', δ)-arcs in α must be k. Thus $stpmult(\alpha) = k$. □

10. Conclusion. The VDAG model proposed in this work can be used to concisely represent hierarchically specified design data in a flexible way that supports design alternatives in engineering design database systems. In fact, there are designs whose representation via VDAGs is exponentially more succinct than is possible with the conventional model of uncontrolled expansion. However, only those design forests which are well structured can be generated via the VDAG paradigm of ancestor based expansion. Problems such as determining whether a forest can be generated by a VDAG are *NP*-complete, even for forests whose heights are bounded. However, for an important class of design forests that include objects from many design applications, namely *UnqStamp*, the problem of determining whether a given forest is generable from a VDAG can be solved in polynomial time. If the answer is "yes," an appropriate VDAG can be generated in polynomial time, although finding a minimum-sized VDAG for a forest in *UnqStamp* is *NP*-hard, even if the height of that forest is bounded.

REFERENCES

[1] D. S. BATORY AND W. KIM, *Modeling concepts for* VLSI CAD *objects*, ACM Trans. Database Systems, 10 (1985), pp. 322–346.

[2] K. R. DITTRICH AND R. A. LORIE, *Version support for engineering database systems*, IEEE Trans. Software Engrg., 14 (1988), pp. 429–437.

[3] C. W. FRASER AND E. W. MYERS, *An editor for revision control*, ACM Trans. Programming Languages and Systems, 9 (1987), pp. 277–295.

[4] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, Freeman, New York, 1979.

[5] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[6] P. HECKEL, *A technique for isolating differences between files*, Comm. ACM, 21 (1978), pp. 264–268.

[7] J. W. HUNT AND M. D. MCILROY, *An Algorithm for Differential File Comparison*, Comput. Sci. Tech. Report 41, AT&T Bell Laboratories, Murray Hill, NJ, 1976.

[8] R. H. KATZ, *A database approach for managing* VLSI *design data*, ACM IEEE 19th Design Automation Conf., Las Vegas, NV, June 1982, pp. 274–282.

[9] ———, *Toward a unified framework for version modeling in engineering databases*, ACM Comput. Surveys, 22 (1990), pp. 375–408.

[10] R. H. KATZ, R. BHATEJA, E. E. CHANG, D. GEDGE, AND V. TRIJANTO, *Design Version Management*, IEEE Design & Test, 4 (1987), pp. 12–22.

[11] R. H. KATZ AND T. J. LEHMAN, *Database support for versions and alternatives of large design files*, IEEE Trans. Software Engrg., SE-10 (1984), pp. 191–200.

[12] D. E. KNUTH, *The Art of Computer Programming*, Vol. 1, 2nd ed., Fundamental Algorithms, Addison-Wesley, Reading MA, 1973.

[13] M. J. ROCHKIND, *The source code control system*, IEEE Trans. Software Engrg., SE-1 (1975), pp. 364–370.

[14] D. G. SEVERANCE AND G. M. LOHMAN, *Differential files: their application to the maintenance of large databases*, ACM Trans. Database Systems, 1 (1976), pp. 256-267.

[15] W. F. TICHY, *RCS—A System for Version Control*, Software—Practice and Experience, 15 (1985), pp. 637-684.

[16] *Unix User's Reference Manual*, Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA, April 1986.

[17] *VHDL Language Reference Manual, Draft Standard* 1076/B, IEEE, May 1987.

[18] L. YU AND D. J. ROSENKRANTZ, *Minimizing Time-Space Cost for Database Version Control*, Acta Inform., 27 (1990), pp. 627-663.

[19] ———, *Ancestor controlled submodule inclusion in design databases*, Proc. 2nd Intl. Conf. on Data and Knowledge Systems for Manufacturing and Engineering, pp. 28-37, Gaithersburg, MD, IEEE, October 1989. IEEE Trans. on Knowledge and Data Engineering, full version to appear.

SIAM J. COMPUT.
Vol. 21, No. 5, pp. 856–862, October 1992

© 1992 Society for Industrial and Applied Mathematics
003

A LOWER BOUND FOR PARALLEL STRING MATCHING*

DANY BRESLAUER† AND ZVI GALIL‡

Abstract. This paper presents an $\Omega(\log\log m)$ lower bound on the number of rounds necessary for finding occurrences of a pattern string $P[1..m]$ in a text string $T[1..2m]$ in parallel using m comparisons in each round. The bound is within a constant factor of the fastest algorithm for this problem [D. Breslauer and Z. Galil, *SIAM J. Comput.*, 19 (1990), pp. 1051–1058] and also holds for an m-processor CRCW-PRAM in the case of a general alphabet. Consequently, the paper derives the parallel complexity of the string matching problem using p processors for general alphabets, which is

- $\Theta(\frac{m}{p})$ if $p \leq \frac{m}{\log\log m}$,
- $\Theta(\log\log m)$ if $\frac{m}{\log\log m} \leq p \leq m$,
- $\Theta(\log\log_{2p/m} p)$ if $m \leq p \leq m^2$,
- $\Theta(1)$ if $p \geq m^2$,

or in short $\Theta(\lceil \frac{m}{p} \rceil + \log\log_{\lceil 1+p/m \rceil} 2p)$.

Key words. string, period, parallel algorithms, lower bounds

AMS(MOS) subject classifications. 68Q10, 68Q20, 68Q25

1. Introduction. The string matching problem is defined as follows: given a string $P[1..m]$ called "pattern" and a string $T[1..n]$ called "text," find all occurrences of the pattern in the text. In the parallel case the output of a string matching algorithm will be a Boolean array $MATCH[1..n]$ that contains a *true* value at each position where an occurrence of the pattern starts. We consider the case where $n = 2m$. The string matching problem has so far defied all attempts of proving lower bounds.

In the sequential case, at a certain time when the problem had a logarithmic space algorithm [9] and linear time algorithm [12], it was conjectured to have a time-space tradeoff [2]. But this conjecture was later disproved when a linear-time constant space algorithm was discovered [10]. Moreover, even a six-head two-way finite automaton can perform string matching in linear time. It is still an open problem as to whether a k-head one-way finite automaton can perform string matching. The only known cases are for $k = 1, 2, 3$ [11], [14], [13], where the answer is negative.

In the parallel case, better and better algorithms have been designed, all on CRCW-PRAM with the weakest form of simultaneous write conflict resolution: all processors which write into the same memory location must write the same value of 1. The best CREW-PRAM algorithms are those obtained from the CRCW algorithms for a logarithmic loss of efficiency. Galil [8] designed an optimal $O(\log m)$ time algorithm for the case of a constant size alphabet. (An optimal algorithm is one with $pt = O(n)$, where t is the time and p is the number of processors used.) Vishkin [17] gave an optimal $O(\log m)$ time algorithm for general alphabet. Breslauer and Galil [4] obtained an optimal $O(\log\log m)$ time algorithm for general alphabet. Recently, Vishkin [18] developed an optimal $O(\log^* m)$ time algorithm. Unlike the cases of the other algorithms, this time bound does not account for the preprocessing of the pattern. The preprocessing

*Received by the editors April 9, 1991; accepted for publication (in revised form) August 5, 1991. This work was partially supported by National Science Foundation grant CCR-90-14605.

†Department of Computer Science, Columbia University, New York, New York 10027. The work of this author was partially supported by an IBM Graduate Fellowship. This work was done while the author was visiting the IBM T. J. Watson Research Center.

‡Department of Computer Science, Columbia University, New York, New York 10027 (galil@cs.columbia.edu) and Department of Computer Science, Tel-Aviv University, Ramat-Aviv, Tel-Aviv 69978, Israel. This work was done while the author was visiting the Laboratoire Informatique Théorique et Programmation, Université Paris 7, Paris, France.

in Vishkin's algorithm takes $O(\log^2 m/\log\log m)$. Vishkin's super fast algorithm raised the question of whether an optimal constant-time algorithm is possible.

In this paper we prove the first lower bound for parallel algorithms that solve the string matching problem. We show that a CRCW-PRAM with m processors requires $\Omega(\log\log m)$ time to perform string matching. Thus, our $O(\log\log m)$ optimal parallel algorithm cannot be improved, and Vishkin's algorithm crucially depends on a slower preprocessing.

As a result we derive the exact parallel complexity of string matching for general alphabets on the CRCW-PRAM. In the case $1 \le p \le m/\log\log m$, it is known to be $\Theta(m/p)$; when $m/\log\log m \le p \le m$ it is $\Theta(\log\log m)$; when $m \le p \le m^2$ it is $\Theta(\log\log_{2p/m} p)$; and when $p \ge m^2$ it is known to be $\Theta(1)$. All these bounds can be summarized in one expression: $\Theta(\lceil m/p\rceil + \log\log_{\lceil 1+p/m\rceil} 2p)$.

Various string matching algorithms use different models. The weakest model is the comparison model in which we test for equality of two symbols. Comparisons of this type are sufficient to solve the string matching problem in the sequential case [3], [12], [10] and in the parallel case [4], [17]. However, although the definition of the string matching problem does not require the alphabet to be ordered, an arbitrary order is used in some algorithms [5], [6]. This assumption is reasonable since the alphabet symbols are encoded numerically, which introduces a natural order. Some other algorithms use a more restricted model where the alphabet symbols are assumed to be encoded as integers in a small range [1], [8]. These algorithms usually take advantage of the fact that the alphabet symbols can be used as indices of an array or that many symbols can be packed together in one register. The case where only comparisons are used is usually referred to in the literature as the general alphabet assumption, while the latter case is usually called fixed alphabet.

Our model is similar to Valiant's parallel comparison tree model [16]. We assume the only access the algorithm has to the input strings is by comparisons that check whether two symbols are equal or not. The algorithm is allowed p comparisons in each round, after which it can proceed to the next round or terminate with the answer. We give a lower bound on the minimum number of rounds necessary in the worst case. We show also that our bound holds even if the algorithm is allowed to perform order comparisons that can result in *less than*, *equal*, or *greater than* answers.

Consider a CRCW-PRAM that solves the string matching problem over a general alphabet. In this case the PRAM can only perform comparisons, but no computation, with its input symbols. Thus, its execution can be partitioned into comparison rounds followed by computation rounds; therefore, a lower bound for the number of rounds in the parallel comparison model immediately translates into a lower bound for the time of the CRCW-PRAM.

If the pattern is given in advance and any preprocessing is free, then this lower bound does not hold, as Vishkin's $O(\log^* m)$ algorithm shows. The lower bound also does not hold for CRCW-PRAM over fixed alphabet strings. Similarly, finding the maximum in the parallel decision tree model has the same lower bound [16], but for small integers the maximum can be found in constant time on a CRCW-PRAM [7].

In §2 we present a lower bound for a related problem of finding the period length of a string. In §3 we show how this lower bound extends to string matching. In §4 we extend our bound to cases where the number of comparisons is greater than m.

2. A lower bound for finding the period of a string. Given a string $S[1..m]$, we say that k is a *period length* of S if $S[i+k] = S[i]$ for $i = 1, \cdots, m-k$. We call k *the period length* of S if it is the minimal period length of S. In this section we prove a lower bound

for the problem of finding the period length of a string $S[1..m]$ using m comparisons in each round. Our lower bound also holds for determining whether such a string has a period of length smaller than $m/2$.

We show a strategy for an adversary to answer $\frac{1}{4}\log\log m$ rounds of comparisons and after which still has the choice of fixing the input string S in two ways: in one the resulting string has a period of length smaller than $m/2$ and in the other it does not have any such period. This implies that any algorithm that terminates in less rounds can be fooled.

We say that an integer k is a possible period length if we can fix S consistently with answers to previous comparisons in such a way that k is a period length of S. For such k to be a period length we need each residue class modulo k to be fixed to the same symbol; thus if $l \equiv j \bmod k$, then $S[l] = S[j]$.

At the beginning of round i the adversary will maintain an integer k_i, which is a possible period length. The adversary answers the comparisons of round i in such a way that some k_{i+1} is a possible period length and few symbols of S are fixed. Let $K_i = m^{1-4^{-(i-1)}}$. The adversary will maintain the following invariants that hold at the beginning of round number i:

(1) k_i satisfies $\frac{1}{2}K_i \leq k_i \leq K_i$.
(2) If $S[l]$ was fixed, then for every $j \equiv l \bmod k_i$, $S[j]$ was fixed to the same symbol.
(3) If a comparison was answered as equal, then both symbols compared were fixed to the same value.
(4) If a comparison was answered as unequal, then
  (a) it was between different residues modulo k_i;
  (b) if the symbols were fixed, then they were fixed to different values.
(5) The number of fixed symbols f_i satisfies $f_i \leq K_i$.

Note that invariants 3 and 4 imply consistency of the answers given so far. Invariants 2, 3, and 4 imply that k_i is a possible period length: if we fix all symbols in each unfixed residue class modulo k_i to a new symbol, a different symbol for different residue classes, we obtain a string consistent with the answers given so far that has a period length k_i.

We start at round number 1 with $k_1 = K_1 = 1$. It is easy to see that the invariants hold initially. We show how to answer the comparisons of round i and how to choose k_{i+1} so that the invariants still hold. All multiples of k_i in the range $\frac{1}{2}K_{i+1} \cdots K_{i+1}$ are candidates for the new k_{i+1}. A comparison $S[l] = S[j]$ must be answered as equal if $l \equiv j \bmod k_{i+1}$. We say that k_{i+1} *forces* this comparison.

THEOREM 2.1 (see [15]). *For large enough n, the number of primes between* 1 *and n denoted by $\pi(n)$ satisfies $\frac{n}{\ln n} \leq \pi(n) \leq \frac{5}{4}\frac{n}{\ln n}$.*

COROLLARY. *The number of primes between $\frac{1}{2}n$ and n is greater than $\frac{1}{4}n/\log n$.*

LEMMA 2.2. *If $p, q \geq \sqrt{m/k_i}$ are relatively prime, then a comparison $S[l] = S[k]$ is forced by at most one of pk_i and qk_i.*

Proof. Assume $l \equiv k \bmod pk_i$, $l \equiv k \bmod qk_i$ for $1 \leq l, k \leq m$. Then also $l \equiv k \bmod pqk_i$. But $pqk_i \geq m$ and $1 \leq l, k \leq m$, which implies $l = k$, a contradiction. □

LEMMA 2.3. *The number of candidates for k_{i+1} that are prime multiples of k_i and satisfy $\frac{1}{2}K_{i+1} \leq k_{i+1} \leq K_{i+1}$ is greater than $K_{i+1}/4K_i \log m$. Each such candidate satisfies the condition of Lemma* 2.2.

Proof. These candidates are of the form pk_i for prime p. The number of such prime values of p can be estimated using the Corollary to Lemma 2.1. It is at least

$$\frac{1}{4}\frac{K_{i+1}}{k_i \log \frac{K_{i+1}}{k_i}} \geq \frac{K_{i+1}}{4K_i \log m}.$$

Each one of these candidates also satisfies the condition of Lemma 2.2 since $k_i \leq K_i$, $pk_i \geq K_{i+1}/2$ and

$$p^2 \geq \frac{1}{k_i}\frac{K_{i+1}^2}{4K_i} = \frac{1}{k_i}\frac{m^{2-2\cdot 4^{-i}}}{4m^{1-4^{-(i-1)}}} = \frac{m}{k_i}\frac{1}{4}m^{2\cdot 4^{-i}} \geq \frac{m}{k_i}. \qquad \square$$

LEMMA 2.4. *There exists a candidate for k_{i+1} in the range $\frac{1}{2}K_{i+1}\cdots K_{i+1}$ that forces at most $4mK_i \log m/K_{i+1}$ comparisons.*

Proof. By Lemma 2.3 there are at least $K_{i+1}/4K_i \log m$ such candidates that are prime multiples of k_i and satisfy the condition of Lemma 2.2. By Lemma 2.2, each of the m comparisons is forced by at most one of them. So the total number of comparisons forced by all these candidates is at most m. Thus, there is a candidate that forces at most $4mK_i \log m/K_{i+1}$ comparisons. □

LEMMA 2.5. *For m large enough and $i \leq \frac{1}{4}\log\log m$, $1 + m^{2\cdot 4^{-i}} 16 \log m \leq m^{3\cdot 4^{-i}}$.*

Proof. For m large enough,

$$\log\log(1 + 16\log m) < \frac{1}{2}\log\log m = \left(1 - \frac{2}{4}\right)\log\log m,$$

$$\log(1 + 16\log m) < 4^{-(1/4)\log\log m}\log m,$$

$$1 + 16\log m < m^{4^{-(1/4)\log\log m}} \leq m^{4^{-i}},$$

from which the lemma follows. □

LEMMA 2.6. *Assume the invariants hold at the beginning of round i and the adversary chooses k_{i+1} to be a candidate which forces at most $4mK_i \log m/K_{i+1}$ comparisons. Then the adversary can answer the comparisons in round i so that the invariants also hold at the beginning of round $i+1$.*

Proof. By Lemma 2.4, such k_{i+1} exists. For each comparison that is forced by k_{i+1} and is of the form $S[l] = S[j]$, where $l \equiv j \bmod k_{i+1}$, the adversary fixes the residue class modulo k_{i+1} to the same new symbol (a different symbol for different residue classes). The adversary answers comparisons between fixed symbols based on the value they are fixed to. All other comparisons involve two positions in different residue classes modulo k_{i+1} (and at least one unfixed symbol) and are always answered as unequal.

Since k_{i+1} is a multiple of k_i, the residue classes modulo k_i split; each class splits into k_{i+1}/k_i residue classes modulo k_{i+1}. Note that if two indices are in different residue classes modulo k_i, then they are also in different residue classes modulo k_{i+1}; if two indices are in the same residue class modulo k_{i+1}, then they are also in the same residue class modulo k_i.

We show that the invariants still hold.

(1) The candidate we chose for k_{i+1} was in the required range.

(2) Residue classes that were fixed before split into several residue classes; all are fixed. Any symbol fixed at this round causes its entire residue class modulo k_{i+1} to be fixed to the same symbol.

(3) Equal answers of previous rounds are not affected since the symbols involved were fixed to the same value by the invariants held before. Equal answers of this round are either between symbols that were fixed before to the same value or are within the same residue class modulo k_{i+1}, and the entire residue class is fixed to the same value.

(4) (a) Unequal answers of previous rounds are between different residue classes modulo k_{i+1} since residue classes modulo k_i split. Unequal answers of this round are between different residue classes because comparisons within the same residue class modulo k_{i+1} are always answered as equal.

(b) Unequal answers that involve symbols that were fixed before this round are consistent because fixed values dictate the answers to the comparisons. Unequal answers that involve symbols that are fixed at the end of this round and at least one that was fixed at this round are consistent since a new symbol is used for each residue class fixed.

(5) We prove inductively that $f_{i+1} \leq K_{i+1}$. We fix at most $4mK_i \log m/K_{i+1}$ residue classes modulo k_{i+1}. There are k_{i+1} such classes and each class has at most $\lceil m/k_{i+1} \rceil \leq 2m/k_{i+1}$ elements. By Lemma 2.5 and simple algebra the number of fixed elements satisfies

$$\begin{aligned} f_{i+1} &\leq f_i + \frac{2m}{k_{i+1}} \frac{4mK_i \log m}{K_{i+1}} \\ &\leq K_i \left[1 + \left(\frac{m}{K_{i+1}} \right)^2 16 \log m \right] \\ &\leq m^{1-4^{-(i-1)}} (1 + m^{2 \cdot 4^{-i}} 16 \log m) \\ &\leq m^{1-4^{-i}} = K_{i+1}. \end{aligned}$$

□

THEOREM 2.7. *Any comparison-based parallel algorithm for finding the period length of a string $S[1..m]$ using m comparisons in each round requires $\frac{1}{4} \log \log m$ rounds.*

Proof. Fix an algorithm that finds the period of S, and let the adversary described above answer the comparisons. After $i = \frac{1}{4} \log \log m$ rounds

$$f_{i+1}, k_{i+1} \leq m^{1-4^{-(1/4) \log \log m}} = \frac{m}{2^{\sqrt{\log m}}} \leq \frac{m}{2}.$$

The adversary can still fix S to have a period length k_{i+1} by fixing each remaining residue class modulo k_{i+1} to the same symbol, different symbols for each class. Alternatively, the adversary can fix all unfixed symbols to different symbols. Note that this choice is consistent with all the the comparisons answered so far by invariants 3 and 4, and the string does not have any period length smaller than $m/2$. Consequently, any algorithm that terminates in less than $\frac{1}{4} \log \log m$ rounds can be fooled. □

THEOREM 2.8. *The lower bound holds also for order comparisons.*

Proof. The adversary gradually defines the linear order of the symbols. He does it in such a way that the answers to comparisons in round i are determined at the round or before. The order is determined by a lexicographic order on a name given to each symbol and is extended for unfixed symbols at each round.

At round i, after k_{i+1} is chosen and before fixing new symbols, all names of unfixed symbols $S[l]$ are extended to be $< l \bmod k_2, l \bmod k_3, \cdots, l \bmod k_{i+1} >$. If a symbol $S[l]$ was fixed at round $j \leq i$, its entire residue class is assigned the name $< l \bmod k_2,$

$l \bmod k_3, \cdots, l \bmod k_{j+1} >$, which is never changed. Note that all other names cannot have this name as a prefix. If two symbols compared at round i are in the same residue class modulo k_{i+1}, then they were fixed at this round or before and given the same name. Symbols that are compared at round i and are in different residue classes modulo k_{i+1} have different names unless they were fixed earlier to the same value. The result of the comparison $(<, =, >)$ is already determined even if the full final name of the symbols is not defined yet. At the end, if the residue classes modulo k_{i+1} are fixed in such a way that k_{i+1} is the period length, then the full names are given as before. If all symbols are fixed to be different, the name given to unfixed position l is $< l \bmod k_2, l \bmod k_3, \cdots, l \bmod k_{i+1}, l >$. □

3. A lower bound for string matching.

THEOREM 3.1. *The lower bound holds also for any comparison-based string matching algorithm.*

Proof. Fix a string matching algorithm. We present to the algorithm a pattern $P[1..m]$ that is $S[1..m]$ and a text $T[1..2m-1]$ that is $S[2..2m]$, where S is a string of length $2m$ generated by the adversary in the way described above. (We use the same adversary that we used in the previous proof; the adversary sees all comparisons as comparisons between symbols in S.) After $\frac{1}{4}\log\log 2m$, rounds, the adversary still has the following choices. He can fix S to have a period length smaller than m, in which case we will have an occurrence of P in T; he can fix all unfixed symbols to completely different characters, which implies that there would be no such occurrence. Thus, the lower bound holds also for any such string matching algorithm. □

4. More comparisons in each round. We can use the trivial algorithm to solve the string matching problem in constant time if m^2 comparisons are available in each round on a CRCW-PRAM; therefore, no more than m^2 processors are necessary. If the number of processors p is smaller than $m/\log\log m$, then we can slow down the $\log\log m$ algorithm in [4] to run in $O(m/p)$ time. We prove below that for $m \le p \le m^2$ the time complexity of the string matching problem becomes $\Theta(\log\log_{2p/m} p)$.

LEMMA 4.1. *If $c > 1$ and $p^{1-c^{-k}} = m/2$, then $k = \Theta(\log\log_{2p/m} p)$.*

Proof. We have $2p/m = p^{c^{-k}}$. Taking logarithms twice we get

$$k = \log_c \frac{\log p}{\log \frac{2p}{m}} = \log_c \log_{2p/m} p. \qquad \square$$

THEOREM 4.2. *The algorithm in* [4] *takes $O(\log\log_{2p/m} p)$ time if we use $m \le p \le m^2$ processors.*

Proof. The algorithm in [4] consists of two phases. The first phase is the pattern analysis phase, which is easily modified to check in round i for period lengths in the range $p^{1-2^{-(i-1)}} \cdots p^{1-2^{-i}}$. This phase terminates when $p^{1-2^{-i}} \ge m/2$, and by Lemma 4.1, $O(\log\log_{2p/m} p)$ rounds are sufficient. The second phase is the text analysis phase, which can also be modified to work within the same time bound. □

THEOREM 4.3. *Any comparison-based parallel algorithm for finding the period length of a string $S[1..m]$ using p comparisons, $m \le p \le m^2$, in each round requires at least $\Omega(\log\log_{2p/m} p)$ rounds.*

Proof. We change m to p in the appropriate places of the proof. In particular, we choose $K_i = p^{1-4^{-(i-1)}}$. The adversary can go on as long as $K_i \le m/2$. By Lemma 4.1, $\Omega(\log\log_{2p/m} p)$ rounds are necessary. □

Acknowledgments. We would like to thank Maxime Crochemore, Raffaele Giancarlo, Roberto Grossi, Pino Italiano, Kunsoo Park, Dominique Perrin, and Neil Sarnak for many helpful discussions and comments on early versions of this paper.

REFERENCES

[1] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin, *Parallel construction of a suffix tree with applications*, Algorithmica, 3 (1988), pp. 347–365.

[2] A. B. Borodin, M. J. Fischer, D. G. Kirkpatrick, N. A. Lynch, and M. Tompa, *A time-space tradeoff for sorting on non-oblivious machines*, Proc. 20th IEEE Symposium on Foundations of Computer Science, 1979, pp. 294–301.

[3] R. S. Boyer and J. S. Moore, *A fast string searching algorithm*, Comm. ACM, 20 (1977), pp. 762–772.

[4] D. Breslauer and Z. Galil, *An optimal $O(\log \log n)$ parallel string matching algorithm*, SIAM J. Comput., 19 (1990), pp. 1051–1058.

[5] M. Crochemore, *String-matching and periods*, Bull. EATCS, October, 1989.

[6] M. Crochemore and D. Perrin, *Two-way string-matching*, J. Assoc. Comput. Mach., 38 (1991), pp. 651–675.

[7] F. E. Fich, R. L. Ragde, and A. Wigderson, *Relations between concurrent-write models of parallel computation*, Proc. 3rd ACM Symposium on Principles of Distributed Computing, 1984, pp. 179–189.

[8] Z. Galil, *Optimal parallel algorithms for string matching*, Inform. and Control, 67 (1985), pp. 144–157.

[9] Z. Galil and J. Seiferas, *Saving space in fast string-matching*, SIAM J. Comput., 2 (1980), pp. 417–438.

[10] ———, *Time-space-optimal string matching*, J. Comput. Syst. Sci., 26 (1983), pp. 280–294.

[11] M. Geréb-Graus and M. Li, *Three one-way heads cannot do string matching*, manuscript, 1990.

[12] D. E. Knuth, J. H. Morris, and V. R. Pratt, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 322–350.

[13] M. Li, *Lower bounds on string-matching*, Tech. Report TR 84-63, Department of Computer Science, Cornell University, Ithaca, NY, 1984.

[14] M. Li and Y. Yesha, *String-matching cannot be done by a two-head one-way deterministic finite automaton*, Inform. Process. Lett., 22 (1986), pp. 231–235.

[15] J. B. Rosser and L. Schoenfeld, *Approximate formulas for some functions of prime numbers*, Illinois J. Math., 6 (1962), pp. 64–94.

[16] L. G. Valiant, *Parallelism in comparison models*, SIAM J. Comput., 4 (1975), pp. 348–355.

[17] U. Vishkin, *Optimal parallel pattern matching in strings*, Inform. and Control, 67 (1985), pp. 91–113.

[18] ———, *Deterministic sampling — a new technique for fast pattern matching*, Proc. 22nd ACM Symposium on Theory of Computation, 1990, pp. 170–179.

SIAM J. COMPUT.
Vol. 21, No. 5, pp. 863–888, October 1992

© 1992 Society for Industrial and Applied Mathematics
004

SEARCHING FOR A MOBILE INTRUDER IN A POLYGONAL REGION*

ICHIRO SUZUKI† AND MASAFUMI YAMASHITA‡

Abstract. The problem of searching for a mobile intruder in a simple polygon by a single mobile searcher is considered. This paper investigates the capabilities of searchers having different degrees of visibility by introducing the searcher having k flashlights whose visibility is limited to k rays emanating from his position, and the searcher having a point light source who can see in all directions simultaneously. This paper presents necessary and sufficient conditions for a polygon to be searchable by various searchers. The paper also introduces a class of polygons for which the searcher having two flashlights is as capable as the searcher having a point light source, and it gives a simple necessary and sufficient condition for such polygons to be searchable by the searcher having two flashlights. The complexity of generating a search schedule under some of these conditions is also discussed. Many of the results are proved using chord systems that represent the visibility relations among the vertices and edges of the given polygon.

Key words. geometry, visibility

AMS(MOS) subject classification. 68E99

1. Introduction. Problems related to visibility inside a simple polygon have been the subject of many recent papers, where two points inside a polygon are said to be mutually visible if the segment between them does not go outside the polygon. Of particular interest to us among these problems is the watchman route problem [3], [4], which is an interesting variation of the well-known art gallery problem of stationing guards in a simple polygon so that every point in the interior of the polygon will be visible from at least one guard [5], [8], [11], [15], [16]. The goal of the watchman route problem is to construct a shortest tour within a given simple polygon so that every point in the interior of the polygon will be visible from at least one point on the tour. Note that this goal can be interpreted as (constructing a path for) finding a *stationary* intruder located in the polygon by a single mobile searcher. The objective of this paper is to consider a dynamic version of the watchman route problem in which the intruder is assumed to be *mobile* rather than stationary. Obviously, if the speed of the intruder is sufficiently large compared to that of the searcher, then even if the searcher can see in all directions simultaneously at any time, depending on the shape of the given polygon it may or may not be possible for the searcher to find the intruder.

Another motivation for considering such a dynamic search problem is the following. The well-known graph search problem [10], [14], [17] of "clearing" an initially "contaminated" graph by using a number of searchers was first introduced as a possible formalization of the following problem [17]: "Suppose a man is lost and wandering unpredictably in a dark cave. A party of searchers who know the structure of the cave is to be sent to find him. What is the minimum number of searchers needed to find the lost man regardless of how he behaves?" If we represent the cave as a polygon (possibly with holes and overlaps) rather than as a graph, then we obtain a geometric search problem that can be considered as a more faithful formalization of the original problem. The problem considered in this paper can be viewed as a restricted version of this geometric problem in which the given polygon is simple and the number of available searchers is one.

*Received by the editors November 28, 1990; accepted for publication (in revised form) September 19, 1991. This work was supported in part by a Scientific Research Grant-in-Aid from the Ministry of Education, Science, and Culture of Japan.

†Department of Electrical Engineering and Computer Science, University of Wisconsin, P.O. Box 784, Milwaukee, Wisconsin 53201 (suzuki@cs.uwm.edu).

‡Department of Electrical Engineering, Faculty of Engineering, Hiroshima University, Kagamiyama, Higashi-Hiroshima 724, Japan (mak@se.hiroshima-u.ac.jp).

We refer to the dynamic search problem informally introduced above as the *polygon search problem*. In this problem we represent both the searcher and the intruder as a point that can move continuously within a given polygon, and assume that the intruder can move arbitrarily faster than the searcher. To investigate the capabilities of the searchers having different degrees of visibility, we introduce the k-searcher for each integer $k \geq 1$ and the ∞-searcher. The k-searcher is the searcher having k "flashlights" whose visibility is limited to k rays emanating from his position, where the directions of the rays can be changed continuously with bounded angular rotation speed. The ∞-searcher is the searcher having the visibility of 360 degrees who can see in all directions simultaneously at any time. Given a simple polygon, the objective here is to decide whether there exists a "schedule" for the given searcher to find the intruder regardless of how he moves, and if so, to generate such a schedule. Detection of a mobile intruder in a simple polygon was first considered in the searchlight scheduling problem [20] in which the rays of stationary "searchlights" are used to find the intruder, but to our knowledge, no concrete results on the polygon search problem have been reported.[1]

We say that a polygon is searchable by a given searcher if there exists a schedule for the searcher. Note that any polygon searchable by the k-searcher is searchable by the k'-searcher for any $k' > k$ and also by the ∞-searcher. In the next section we show that (1) the 2-searcher is strictly more capable than the 1-searcher (i.e., there exists a polygon searchable by the 2-searcher but not by the 1-searcher), and (2) any n-sided polygon searchable by the ∞-searcher is searchable by the $(\lfloor n/2 \rfloor + 1)$-searcher. Thus a fundamental question here is whether there exists an n-sided polygon searchable by the k-searcher but not by the $(k-1)$-searcher for some $3 \leq k \leq \lfloor n/2 \rfloor + 1$. So far we have not been able to find such a polygon, and we conjecture that any polygon searchable by the ∞-searcher is actually searchable by the 2-searcher. One of the goals of this paper is to discuss a case in which the 2-searcher is in fact equally capable as the ∞-searcher.

The main results of this paper are the following. First, we present simple necessary conditions and sufficient conditions for a polygon to be searchable by various searchers. Specifically, we show that no polygon P searchable by the 1-searcher can have three points x, y, and z such that no point in $\pi(x,y)$ is visible from z, no point in $\pi(y,z)$ is visible from x and no point in $\pi(z,x)$ is visible from y, where for points u and v in P, $\pi(u,v)$ denotes the Euclidean shortest path between u and v within P. We prove a similar result for the ∞-searcher using the concept of "2-visibility" instead of visibility, where a point u is said to be 2-visible from a point v if u is visible from a point visible from v. The sufficient conditions state that a polygon is searchable (1) by the 1-searcher if it is weakly visible from an edge, (2) by the 2-searcher if it is weakly 2-visible from an edge, and (3) by the 2-searcher if it is weakly visible from $\pi(u,v)$ for some points u and v on the boundary, where a region Q is said to be weakly visible (or weakly 2-visible) from a region R if every point in Q is visible (or 2-visible) from some point in R. The complexity of generating search schedules under these conditions is also discussed. Next, we introduce a structure called *chord system* that represents the visibility relations among the vertices and edges of a polygon, and we discuss its basic properties. We then use chord systems to show that any polygon searchable by the 1-searcher (or the ∞-searcher) must be weakly visible (or weakly 2-visible) from $\pi(u,v)$ for some vertices u and v. Finally, we investigate the case in which the given polygon is a *hedgehog*, which is a simple polygon consisting of a convex body and narrow hooked pins (corridors) (see Figs. 3, 5, and 6). We show that any hedgehog searchable by the ∞-searcher is also searchable by the 2-searcher, and that a

[1] Recently we found that the polygon search problem is mentioned as an open problem and called the *hunting problem* in [19].

schedule of the 2-searcher of length $O(m)$ for searching a given n-sided hedgehog having m hooked pins can be generated in $O(n + m \log n)$ time. The results on hedgehogs are again obtained using chord systems.

The polygon search problem is stated formally in §2. Simple necessary conditions and sufficient conditions are presented in §3. Chord systems and their basic properties are discussed in §4. Section 5 presents additional necessary conditions that are obtained using chord systems. The case when the given polygon is a hedgehog is discussed in §6. Concluding remarks are found in §7.

2. Problem formulation. Let P be a simple polygon. We denote by ∂P the boundary of P. Two points x and $y \in P$ are said to be mutually *visible* if $\overline{xy} \subseteq P$. We let $V(x)$ denote the set of points in P that are visible from x.

Let us introduce the searcher who can walk within P holding k flashlights, where k is a positive integer, and a flashlight emits a single light beam that is blocked as soon as it intersects with the exterior of P. We call this searcher the *k-searcher*.

DEFINITION 1. A *schedule* of the k-searcher for P is a tuple $\mathcal{S} = \langle S, F_1, F_2, \cdots, F_k \rangle$ of $k+1$ continuous functions $S : [0, T] \to P$ and $F_1, F_2, \cdots, F_k : [0, T] \to \mathcal{R}$, where $[0, T]$ is an interval of real time and $\mathcal{R}$ is the set of real numbers. A point $x \in P$ is *illuminated* at time $t \in [0, T]$ during the execution of $\mathcal{S}$ if x is on the intersection of $V(S(t))$ and the k rays emanating from $S(t)$ in the directions $F_1(t), F_2(t), \cdots, F_k(t)$, respectively.

$S(t)$ is the location of the searcher and $F_1(t), F_2(t), \cdots, F_k(t)$ are the directions of the k flashlights at time t, respectively, where directions are measured in radian counterclockwise from the positive x-axis. Intuitively, the illuminated points are those that the k-searcher can "see" at any given time. Note that the visibility of the k-searcher is limited to k rays emanating from his position.

Let us introduce another type of searcher who can see in all directions simultaneously. We call this searcher the *∞-searcher*, since we can imagine that this searcher has a point light source that can be viewed as a collection of infinitely many flashlights aimed in all directions.

DEFINITION 2. Let P be a simple polygon. A *schedule* of the ∞-searcher for P is $\mathcal{S} = \langle S \rangle$, where $S : [0, T] \to P$ is any continuous function over an interval $[0, T]$ of real time. A point $x \in P$ is said to be *illuminated* at time $t \in [0, T]$ during the execution of $\mathcal{S}$ if $x \in V(S(t))$.

$S(t)$ is the location of the ∞-searcher at time t. Again, the illuminated points are those that the ∞-searcher can "see" at any given time. Note that the set of illuminated points at time t coincides with the set of points that are visible from $S(t)$.

DEFINITION 3. Let $\mathcal{S}$ be a schedule of any searcher defined over the interval $[0, T]$ of real time. A point $x \in P$ is said to be *contaminated* at time $t \in [0, T]$ during the execution of $\mathcal{S}$ if there exists a continuous function $I : [0, t] \to P$ such that for any $t' \in [0, t]$, $I(t')$ is not illuminated at time t'. A point that is not contaminated is said to be *clear*. A region $R \subseteq P$ is said to be *contaminated* if it contains a contaminated point; otherwise, it is *clear*.

The function I in Definition 3 represents a path of the intruder who can move continuously with unbounded speed. Intuitively, $x \in P$ is contaminated at time $t \in [0, T]$ if and only if the intruder can reach x at t without being seen by the searcher in the interval $[0, t]$. Note that at time zero, a point is contaminated if and only if it is not illuminated. Also, at any time an illuminated point is clear, and a contaminated point remains contaminated until it becomes illuminated.

DEFINITION 4. Let $\mathcal{S}$ be a schedule of the k-searcher (or ∞-searcher) for P defined over the interval $[0, T]$ of real time. $\mathcal{S}$ is called a *search schedule* for P if P is clear at time

T. P is said to be *k-searchable* (or *∞-searchable*) if there exists a search schedule of the k-searcher (or ∞-searcher) for P.

Since the "speed" at which the searcher "executes" a given schedule is irrelevant to whether or not P is clear at the end of the execution, we represent a schedule $\mathcal{S}$ of any searcher simply as a sequence of some elementary actions, instead of giving formal descriptions of the functions contained in $\mathcal{S}$. The only elementary action of the ∞-searcher that we use here is to move over a given line segment, and hence a schedule of the ∞-searcher is simply given as a directed polygonal chain within P. For the k-searcher, the elementary actions we use are (1) to aim a flashlight at a given point, (2) to rotate a flashlight either clockwise or counterclockwise to illuminate a given point, (3) to move over a segment aiming the flashlights at or through given points, and (4) to move over a segment aiming the flashlights in given fixed directions. (All the schedules we develop require these elementary actions to be executed one at a time.) Obtaining a formal description of a schedule as a collection of functions from a sequence of such actions is straightforward. We define the *length* of a schedule to be the number of elementary actions it contains.

Example 1. Figure 1 shows a simple polygon and the directed polygonal chain $\overline{abcde}$ determined by a search schedule of the ∞-searcher. The chain consists of four segments, and hence the length of the schedule is four. The regions that are clear when the ∞-searcher reaches a, b, c, d, and e are shown shaded in Figs. 1(a), (b), (c), (d), and (e), respectively (the ∞-searcher is shown by •).

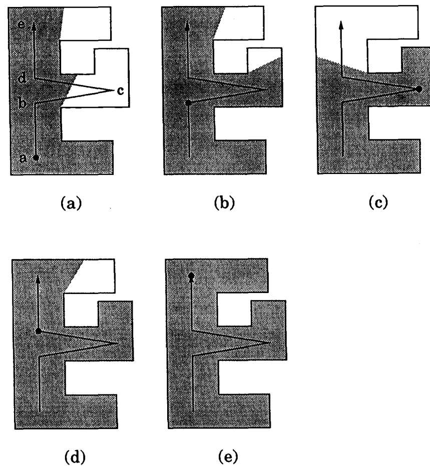


FIG. 1. *A polygon and a search schedule of the ∞-searcher.*

Example 2. Figure 2 shows how the polygon of Example 1 can be cleared by the 2-searcher. Again, clear regions are shown shaded. Initially the two flashlights are aimed at a, and then one of them is rotated counterclockwise until it is aimed at b (Fig. 2(a)). Then the 2-searcher moves to a new position aiming the two flashlights continuously at a and b, respectively (Fig. 2(b)). This movement requires two consecutive moves over a segment, since the positions of the 2-searcher in Figs. 2(a) and (b) are not mutually visible. The rest of the operation should be self-explanatory. Note that only the elementary actions introduced above are used.

Some points in the polygon may have to be recontaminated repeatedly during the search, as the following example shows.

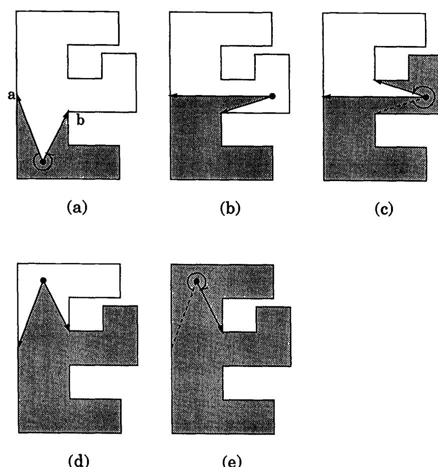


FIG. 2. *Clearing the polygon of Example 1 using the 2-searcher.*

Example 3. The polygon shown in Fig. 3 consists of a square and seven narrow hooked pins, where each pin is drawn schematically as a polygonal chain. Each pin is assumed to be sufficiently narrow so that the visibility (into the square) from the points near the bend is limited approximately to the neighborhood of the corresponding dotted line. Let $a, b, \cdots$, and g be the vertices at the "tip" of the pins, as is shown in the figure. Note that the ∞-searcher must visit each of $V(a)$, $V(b), \cdots$, and $V(g)$ to clear these tips. This polygon is ∞-searchable, since it can be cleared by the ∞-searcher who visits $V(a)$, $V(b)$, $V(g)$, $V(c)$, $V(d)$, $V(a)$, $V(e)$, $V(f)$, $V(c)$, $V(g)$, $V(b)$, and $V(a)$ in this order by using the shortest paths to go from one region to the next. (An initial segment of this path is shown in Fig. 3.) Table 1 shows the clear tip vertices at the moment each of the tip vertices are cleared. Note that a is recontaminated twice during the search. This is unavoidable, as explained below. We may assume that the ∞-searcher starts the search from a point in $V(a)$, since if a tip vertex x other than a is cleared first among all the tip vertices, then x becomes contaminated when any other tip vertex y is cleared next, and hence a will be the only clear tip vertex when the ∞-searcher eventually reaches $V(a)$. Also, we may assume that the search ends at a point in $V(a)$, since the "reversal" of any search schedule is also a search schedule.[2] Now, note that at the moment he enters $V(d)$ or $V(e)$ for the first time after leaving $V(a)$, at least four tip vertices including a are contaminated. (For example, if the ∞-searcher is in $V(d)$, then at least a, e, f and g are contaminated.) If a is recontaminated only once during the search, then the ∞-searcher must clear all contaminated tip vertices other than a before he enters $V(a)$. But this is not possible, since as soon as he clears a contaminated tip vertex x other than a, x becomes the only clear tip vertex. Therefore, a must be recontaminated at least twice.

Example 4. The polygon shown in Fig. 3 can also be cleared by the 2-searcher. Basically, the 2-searcher clears the tip vertices in the same order as the ∞-searcher, and uses the two flashlights appropriately so that the set of clear tip vertices at any given moment is the same as that at the corresponding moment during the execution of the schedule

[2]For any function $f : [0,T] \to P$, its reversal is the function $f^{-1} : [0,T] \to P$ such that $f^{-1}(t) = f(T-t)$ for all $t \in [0,T]$. Obviously, if the ∞-searcher given by S cannot detect the intruder given by I, then the ∞-searcher given by S^{-1} cannot detect the intruder given by I^{-1}. Thus S is a search schedule if and only if S^{-1} is a search schedule.

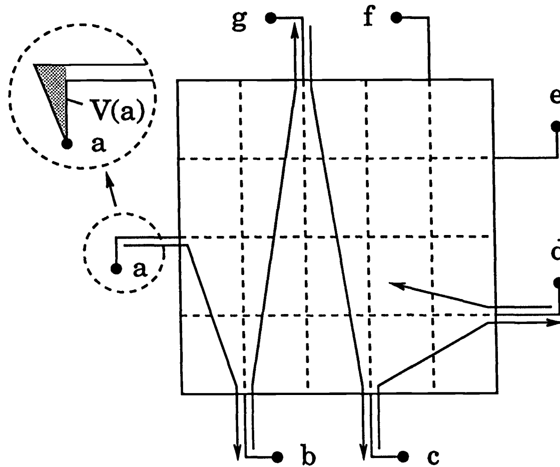


FIG. 3. *A square with seven hooked pins.*

TABLE 1
Clear tip vertices at the moment each tip vertex is cleared.

Tip vertex just cleared	Clear tip vertices at that moment
a	a
b	a, b
g	a, b, g
c	a, b, c, g
d	b, c, d
a	a, b, c, d
e	a, b, c, d, e
f	d, e, f
c	c, d, e, f
g	c, d, e, f, g
b	b, c, d, e, f, g
a	a, b, c, d, e, f, g

for the ∞-searcher. Specifically, the 2-searcher starts from a point in $V(a)$ sweeping the boundary of P by the "left" and "right" flashlights (called F_L and F_R, respectively). See Fig. 4 for illustration. When b is cleared for the first time, the 2-searcher is in $V(b)$ aiming F_R at b and F_L at a point in the boundary ∂P between a and g, so that a and b are clear. Similarly, when g is cleared for the first time, the 2-searcher is in $V(g)$ aiming F_L at g and F_R at a point in ∂P between b and c, so that a, b, and g are clear. Tip vertex c is cleared in a similar manner. When he moves from $V(c)$ to $V(d)$, F_R is advanced to d and F_L is moved backward (from a point in ∂P between f and g) to a point in ∂P between a and b, so that b, c, and d are clear. The rest of the schedule is similar, and we leave details to the reader.

Example 5. Figures 5 and 6 show schematic drawings of two polygons that are not ∞-searchable. The one shown in Fig. 5 consists of a rectangle and three narrow hooked pins, where again each pin is drawn as a polygonal chain with a bend. To see that this polygon is not ∞-searchable, note that whenever one of the three tips a, b, and c is illuminated by the ∞-searcher, there exists a path between the remaining two tips not

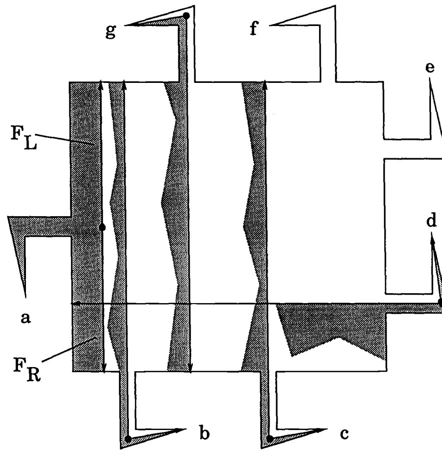


FIG. 4. *Clearing the polygon of Example 3 using the 2-searcher.*

containing any illuminated point. This, together with the fact that initially at most one of the three tips can be illuminated, implies that at most one of the three tips can be clear at any time during the search by the ∞-searcher. Thus this polygon is not searchable by the ∞-searcher. The polygon shown in Fig. 6 is a "windmill" consisting of a convex body and five narrow hooked pins. Note that initially at most one of the five tips a, b, c, d, and e can be clear. Thus, without loss of generality, let a be the tip that becomes clear first among the five. If any of b, c, and d is cleared next, then that tip will be the only clear tip since a becomes recontaminated. It is possible to clear e without recontaminating a, but both become recontaminated if b or c is cleared next, and d and e will be the only clear tips if d is cleared next. Therefore, at most two of the five tips can become clear simultaneously, and hence there exists no search schedule for this polygon.

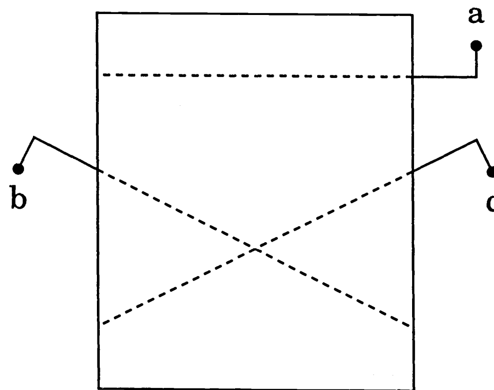


FIG. 5. *A polygon which is not ∞-searchable.*

The following proposition is immediate from definition.

PROPOSITION 1. *Any simple polygon that is k-searchable is k′-searchable for any $k' > k$ and ∞-searchable.*

THEOREM 1. *There exists a simple polygon which is 2-searchable but not 1-searchable.*

Proof. We have already seen in Example 2 that the polygon shown in Fig. 7 is 2-searchable. Points a, b, and c shown in the figure have the property that whenever one

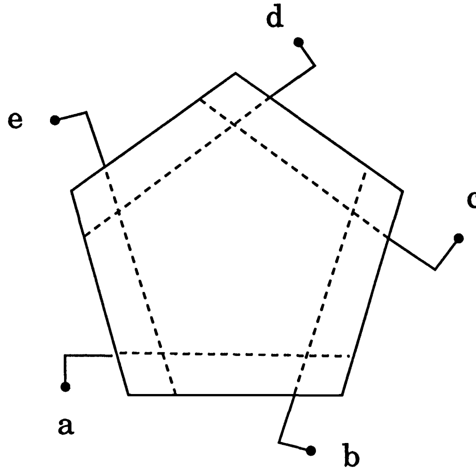


FIG. 6. *Another polygon which is not* ∞*-searchable.*

of them is illuminated by the 1-searcher, there exists a path between the remaining two points not containing any illuminated point. This, together with the fact that initially at most one of the three points can be illuminated, implies that at most one of the three points can be clear at any time during the search by the 1-searcher. Thus the polygon is not searchable by the 1-searcher. □

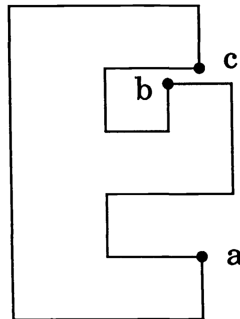


FIG. 7. *A polygon which is not* 1*-searchable.*

THEOREM 2. *If an* n*-sided simple polygon is* ∞*-searchable, then it is* $(\lfloor n/2 \rfloor + 1)$*-searchable.*

Proof. We only give an outline of the proof. Let P be an n-sided simple polygon P that is ∞-searchable. If P is star-shaped [12], then it can be cleared by the 2-searcher stationed in the kernel who (1) aims both flashlights at an arbitrary point on the boundary of P and then (2) rotates one of the flashlights clockwise for 2π without rotating the other. Now assume that P is not star-shaped. Let $S : [0, T] \to P$ be a path of the ∞-searcher for clearing P. First, we place the $(\lfloor n/2 \rfloor + 1)$-searcher at $S(0)$ and aim one flashlight at every reflex vertex which blocks visibility from $S(0)$, as is shown in Fig. 8. Note that the number of such reflex vertices is at least one (since P is not star-shaped) and at most $\lfloor n/2 \rfloor$. Next, we clear $V(S(0))$ by aiming another flashlight at any of the reflex vertices blocking visibility from $S(0)$ and rotating it clockwise for 2π. When this is done, basically we move the $(\lfloor n/2 \rfloor + 1)$-searcher along the path determined by S in

such a way that one flashlight is aimed at every reflex vertex which blocks visibility from his position. (When the $(\lfloor n/2 \rfloor + 1)$-searcher reaches a reflex vertex v that has been blocking visibility, he must stay at v temporarily and perform additional rotations of the flashlights to clear the points in $V(v)$ that have not been visible. We leave details to the reader.) This guarantees that, when the $(\lfloor n/2 \rfloor + 1)$-searcher is at x, the intruder can move (without being seen) only within a single maximal connected region of $P - V(x)$ not visible from x. Therefore, if it were possible for the intruder to avoid being seen by the $(\lfloor n/2 \rfloor + 1)$-searcher until the end of the execution of this schedule, then it would also be possible for him to avoid being seen by the ∞-searcher who moves along the path determined by S. This is a contradiction. □

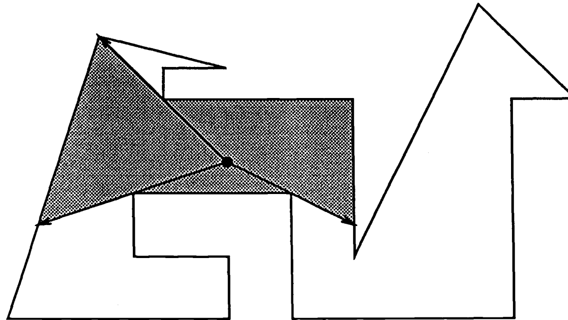


FIG. 8. *Simulation of the ∞-searcher by the $(\lfloor n/2 \rfloor + 1)$-searcher.*

Whether there exists an n-sided simple polygon that is k-searchable but not $(k-1)$-searchable for some $3 \leq k \leq \lfloor n/2 \rfloor + 1$ is an interesting question. So far we have not been able to find such a polygon, and we conjecture that any ∞-searchable polygon is actually 2-searchable. In §6 we show that this indeed is the case if the given polygon has a certain topology.

3. Simple necessary conditions and sufficient conditions. Let P be a simple polygon. For a point $x \in P$, define $V^2(x) = \bigcup_{y \in V(x)} V(y)$. Note that $V(x) \subseteq V^2(x)$. If $y \in V^2(x)$, then we say that y is *2-visible* from x. For regions Q and $R \subseteq P$, we say that Q is *weakly visible* (or *weakly 2-visible*) from R if every point in Q is visible (or 2-visible) from some point in R. (In this definition, if R consists of a single point p, then we simply say that Q is visible (or 2-visible) from p.) For points x, y, and $z \in P$, y and z are said to be *separable* (or *2-separable*) by x if every path within P between y and z contains at least one point in $V(x)$ (or $V^2(x)$). Note that since P is simple, y and z are separable (or 2-separable) by x if and only if $\pi(y, z)$ contains at least one point in $V(x)$ (or $V^2(x)$), where $\pi(y, z)$ is the Euclidean shortest path within P between y and z. See Fig. 9 for an illustration of these concepts.

Points x, y, and $z \in P$ are said to be *mutually nonseparable* (or *mutually non–2-separable*) if no two points out of the three are separable (or 2-separable) by the third. The proofs of the following two theorems are essentially similar to that of Theorem 1.

THEOREM 3. *Let P be a simple polygon. If P is 1-searchable, then no three points in P are mutually nonseparable.*

Proof. Suppose that there are three points in P that are mutually nonseparable. Then (1) at most one of them can be clear at the beginning of the schedule of the 1-searcher, and (2) whenever one of them is illuminated by the 1-searcher, there exists a path between the remaining two points not containing any illuminated point. Thus at

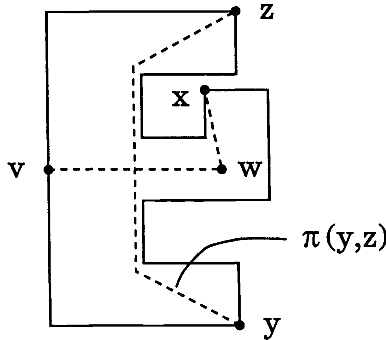


FIG. 9. *w is visible from x; v is 2-visible from x; y and z are separable by w and 2-separable by x.*

most one of the three points can be clear at any time, and hence P cannot be cleared by the 1-searcher. □

THEOREM 4. *Let P be a simple polygon. If P is ∞-searchable, then no three points in P are mutually non–2-separable.*

Proof. Suppose that there are three points in P that are mutually non–2-separable. Then (1) at most one of them can be clear at the beginning of the schedule of the ∞-searcher, and (2) whenever one of them is illuminated by the ∞-searcher, there exists a path between the remaining two points not containing any illuminated point. Thus at most one of the three points can be clear at any time, and hence P cannot be cleared by the ∞-searcher. □

By Theorem 4, the polygon shown in Fig. 5 is not ∞-searchable, since tips a, b, and c are mutually non–2-separable. On the other hand, the non-∞-searchable polygon shown in Fig. 6 has no three points which are mutually non–2-separable. The non–∞-searchability of this polygon follows from Theorem 13 given in §5.

Now we present some sufficient conditions for a polygon to be searchable and discuss related complexity issues.

THEOREM 5. *Let P be a simple polygon that is weakly visible from an edge $\overline{uv}$. Then P is 1-searchable. Furthermore, P can be cleared by the 1-searcher who moves only within $\overline{uv}$.*

Proof. Let $u = p_1, p_2, \cdots, p_n = v$ be the clockwise listing of the vertices of P. For convenience we assume that the searcher is located on $\overline{uv}$ facing the interior of P, and hence u is to his left and v is to his right. For each p_i, $2 \le i \le n$, let $\mu(p_i)$ (or $\nu(p_i)$) be the point closest to u (or v) on $\overline{uv}$ that is visible from p_i. We denote by S the position of the 1-searcher. P can be cleared by the 1-searcher as follows. Initially, the 1-searcher is located at u aiming the flashlight F at p_2. At this moment $S = u$ and the region to the left of $\overline{Sp_2}$ is clear. Suppose that for some $2 \le i \le n-1$, the 1-searcher is located at S, p_i is visible from S, and the region to the left of $\overline{Sp_i}$ is clear. There are three possible movements of the 1-searcher for clearing p_{i+1}.

1. If S is strictly to the left of $\mu(p_{i+1})$, then the 1-searcher moves right from S to $\mu(p_{i+1})$ aiming F at p_i. Note that the triangle $Sp_i\mu(p_{i+1})$ lies within P, and hence p_i is visible from every point in $\overline{S\mu(p_{i+1})}$, as is shown in Fig. 10(a). Thus when the 1-searcher reaches $\mu(p_{i+1})$, the region to the left of $\overline{Sp_{i+1}}$ is clear, where S is now at $\mu(p_{i+1})$.
2. If S is between $\mu(p_{i+1})$ and $\nu(p_{i+1})$, inclusive, then edge $\overline{p_ip_{i+1}}$ is visible from S. Thus the 1-searcher simply rotates F clockwise from p_i to p_{i+1} and clears the region to the left of $\overline{Sp_{i+1}}$. See Fig. 10(b).

3. If S is strictly to the right of $\nu(p_{i+1})$, then $\overline{Sp_i}$ and $\overline{p_{i+1}\nu(p_{i+1})}$ intersect. See Fig. 10(c). Let r be the intersection. Since triangles $\nu(p_{i+1})rS$ and $p_i r p_{i+1}$ both lie within P, when the 1-searcher moves left aiming F through r and reaches $\nu(p_{i+1})$, the region to the left of $\overline{Sp_{i+1}}$ becomes clear, where S is now at $\mu(p_{i+1})$.

P becomes clear when $p_n = v$ is cleared. □

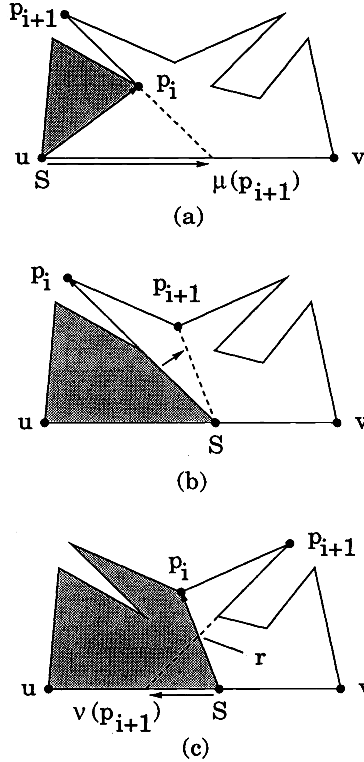


FIG. 10. *Illustration for the proof of Theorem* 5.

Since the points $\mu(p_i)$ and $\nu(p_i)$ for all i can be computed in $O(n)$ time by a method given in [1], a search schedule based on the discussion given in the proof of Theorem 5 can be generated in $O(n)$ time. The length (i.e., the number of elementary actions) of the resulting schedule of the 1-searcher is $n-1$. We also note that given a simple polygon P, all edges of P from which P is weakly visible can be found in linear time [18].

THEOREM 6. *Let P be a simple polygon that is weakly 2-visible from an edge $\overline{uv}$. Then P is 2-searchable.*

Proof. We only give an outline of the proof. Let P' be the subpolygon of P consisting of the points that are weakly visible from $\overline{uv}$. Then P' is 1-searchable by Theorem 5, and any maximal region $Q \subseteq P - P'$ not weakly visible from $\overline{uv}$ is weakly visible from its "lid" (i.e., the boundary between P' and Q). Thus the 2-searcher can clear P by executing the search schedule for P' using flashlight F_1, in such a way that whenever the flashlight illuminates the lid $\overline{xy}$ of such Q, where u, x, y and v occur clockwise in ∂P, he (1) moves straight from his position S to x aiming F_1 and the other flashlight F_2 at x and S, respectively (Fig. 11(a)), (2) clears Q from $\overline{xy}$ using F_1 by the method given in the proof of Theorem 5 aiming F_2 continuously at S so that the intruder will not move across $\overline{Sx}$ (Fig. 11(b)), and then (3) returns to S aiming F_1 and F_2 at y and S, respectively (Fig. 11(c)). □

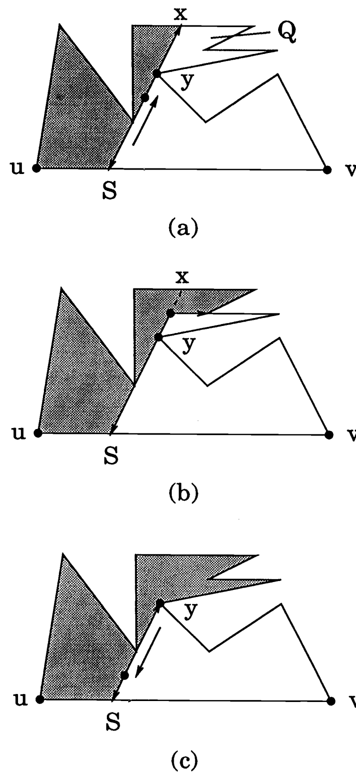


FIG. 11. *Illustration for the proof of Theorem* 6.

Search schedules described in the proof of Theorem 6 can be generated in linear time given a triangulation of P, since (1) the regions Q not weakly visible from $\overline{uv}$ can be found in sorted order in linear time by basically constructing the weak visibility polygon [7] (consisting of the points weakly visible from $\overline{uv}$) from the triangulation, and (2) a schedule for clearing each Q can be generated in time linear to the size of Q. A triangulation of a simple polygon can be obtained in linear time [2].

Theorems 5 and 6 can easily be extended to the case in which P is weakly (2-)visible from a consecutive portion of the boundary of P that is convex toward either the interior or exterior of P. We omit the details to save space.

THEOREM 7. *Let P be a simple polygon that is weakly visible from $\pi(u, v)$ for some points u and $v \in \partial P$. Then P is 2-searchable.*

Proof. We only give an outline of the proof. See Fig. 12 for illustration. Basically, the 2-searcher moves from u to v along $\pi(u, v)$ aiming the "left" and "right" flashlights (called F_L and F_R, respectively) in opposite directions perpendicular to the direction of his movement. Whenever a flashlight, say F_L, illuminates a reflex vertex z blocking visibility from his position x, he does the following aiming the other flashlight (F_R in this case) through x continuously: (1) Move straight to z, (2) clear the region Q behind z not visible from x by F_L from the "lid" of Q using the method given in the proof of Theorem 5 (this is possible since Q is weakly visible from $\pi(u, v)$, and hence 2-visible from x), and (3) return straight to x aiming F_L at or through z. (Before the 2-searcher starts to move from u, the regions, if any, "behind" the rays of F_L and F_R must be cleared by executing

the above operation regarding u as both x and z. Similar operations may be needed to complete the search when he reaches v.) When he reaches a corner $y \in \partial P$ of $\pi(u, v)$ at which $\pi(u, v)$ makes a right (or left) turn, he sweeps the opposite boundary of P using F_L (or F_R) until its ray becomes perpendicular to the direction in which he moves next, clearing all regions not visible from y using a technique similar to the one described above (this is possible since the portion of the boundary to be cleared is 2-visible from y). Certainly P is cleared by this method. □

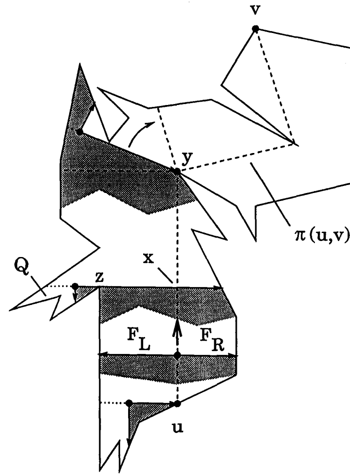


FIG. 12. *Illustration for the proof of Theorem* 7.

For an n-sided polygon P, points u and v satisfying the condition of Theorem 7 can be found in $O(n \log n)$ time (if they exist). The basic idea is similar to that given in [9] for finding a segment within P from which P is weakly visible. To save space, we only give an outline. We construct a set C of chords of P[3] having the property that P is weakly visible from $\pi(u, v)$ for some u and $v \in \partial P$ if and only if there exist u' and $v' \in \partial P$ such that $\pi(u', v')$ intersects every chord in C. Then we map the chords in C to a set C' of chords of a unit circle[4] preserving the order in which their endpoints appear in ∂P. We can then show that there exist u and $v \in \partial P$ such that $\pi(u, v)$ intersects every chord in C if and only if there exists a stabbing line that intersects every chord in C'. As is explained in [9], the set C (of size $O(n)$) can be constructed in $O(n \log n)$ time using the bullet shooting algorithm of [7] and a modified version of an algorithm given in [21]. (*Bullet shooting* is the problem of finding the first point in ∂P hit by the ray emanating from a given point in P in a given direction.) The set C' is constructed from C in linear time. A stabbing line for C' can be found (if it exists) in $O(n \log n)$ time by using an algorithm of [6], and a pair of u and v can be found in $O(\log n)$ time from the given stabbing line using binary search over the intervals of the circumference of the unit circle determined by the endpoints of the chords. Thus the total time needed is $O(n \log n)$. We leave the details to the reader.

Given such u and v, the search schedule described in the proof of Theorem 7 can be generated in additional $O(n \log n)$ time. First, we compute $\pi(u, v)$ in linear time from a triangulation of P [13]. Once $\pi(u, v)$ is computed, we can find all the regions (called Q in the proof) not visible from the 2-searcher and those region of P that must be cleared

[3] A *chord* of P is a line segment within P whose endpoints are in ∂P.

[4] A *chord* of a circle is a line segment whose endpoints are on its circumference.

from corners (called y in the proof) of $\pi(u, v)$ in $O(n \log n)$ time using the bullet shooting algorithm of [7]. Since a schedule for each such region can be found in time linear to its size as we discussed above, the total time needed to generate a schedule is $O(n \log n)$. The total number of elementary actions is clearly $O(n)$.

By Proposition 1, any polygon P satisfying the condition of Theorem 7 is ∞-searchable. In fact, we can show that the ∞-searcher can clear it by simply moving from u to v along $\pi(u, v)$. To save space, we leave the details to the reader.

4. Chord systems. In this section we introduce *chord systems*. Chord systems are used extensively in §5 to prove additional necessary conditions for a polygon to be searchable, and also in §6 to study a class of polygons called hedgehogs. We start with some definitions.

Let $a_1, b_1, a_2, b_2, \cdots, a_n, b_n$ be distinct points that appear in this order clockwise on the circumference of a unit circle. Let $A = \{a_1, a_2, \cdots, a_n\}$ and $B = \{b_1, b_2, \cdots, b_n\}$. For any sets X and $Y \subseteq A \cup B$, $X \times Y$ denotes the set of chords of the unit circle connecting a point in X and a point in Y. A chord c of the unit circle is said to *separate* two disjoint sets A_1 and $A_2 \subseteq A$ if $c \cap \overline{x_1x_2} \neq \emptyset$ for any chord $\overline{x_1x_2} \in A_1 \times A_2$. If c separates $\{a_i\}$ and $\{a_j\}$, then we simply say that c separates a_i and a_j.

DEFINITION 5. A set $\mathcal{C} \subseteq A \times B$ of chords is called a *chord system* if for any $a \in A$ there exists some $b \in B$ such that $\overline{ab} \in \mathcal{C}$. The sets A and B are called the *base sets* of $\mathcal{C}$.

DEFINITION 6. Let $\mathcal{C} \subseteq A \times B$ be a chord system. Let A_1 and $A_2 \subseteq A$ be two disjoint sets. We say that a point $a \in A$ *separates* A_1 and A_2 if for any two points $x_1 \in A_1$ and $x_2 \in A_2$ there exists a chord $\overline{ab} \in \mathcal{C}$ which separates them. We say that a point $a \in A$ *strongly separates* A_1 and A_2 if there exists a chord $\overline{ab} \in \mathcal{C}$ that separates every pair $x_1 \in A_1$ and $x_2 \in A_2$ of points.

The separability and the strong separability are equivalent for singleton sets. If a (strongly) separates $\{a_i\}$ and $\{a_j\}$, then we simply say that a (strongly) separates a_i and a_j. Note that for any $a_i, a_j \in A$, a_i strongly separates a_i and a_j.

We use chord systems to represent the visibility relations among the vertices and edges of the given polygon. Let P be an n-sided polygon having vertices $p_1, p_2, \cdots, p_n$ and edges $e_1, e_2, \cdots, e_n$, where the vertices are taken clockwise and for each $1 \leq i \leq n$, e_i is the edge between p_i and p_{i+1}.[5] We define chord systems $\mathcal{C}_1(P)$ and $\mathcal{C}_2(P)$ with base sets $A = \{p_1, p_2, \cdots, p_n\}$ and $B = \{e_1, e_2, \cdots, e_n\}$ as follows[6]:

1. $\overline{p_ie_j} \in \mathcal{C}_1(P)$ if and only if some point in e_j is visible from p_i;
2. $\overline{p_ie_j} \in \mathcal{C}_2(P)$ if and only if some point in e_j is 2-visible from p_i.

It is immediate from the definition that vertices p_i and p_j are separable by p_k in P if and only if points p_i and p_j are separated by p_k in $\mathcal{C}_1(P)$, and vertices p_i and p_j are 2-separable by p_k in P if and only if points p_i and p_j are separated by p_k in $\mathcal{C}_2(P)$.[7] $\mathcal{C}_1(P)$ and $\mathcal{C}_2(P)$ are similar in spirit to a circular embedding of a visibility graph described in [16].

Example 6. The chords incident on p_1 in $\mathcal{C}_1(P)$ are drawn as solid curved paths within P in Fig. 13. In $\mathcal{C}_2(P)$, an additional two chords drawn as broken curved paths are also incident on p_1. The reader can verify that $\mathcal{C}_2(P) = \{\overline{p_ie_j} \mid 1 \leq i, j \leq 8\} - \{\overline{p_1e_5}, \overline{p_2e_5}, \overline{p_5e_1}, \overline{p_6e_1}\}$.

We say that a vertex x of P is *hit* at time t if x is illuminated at t but not illuminated in the interval $[t - \delta, t)$ for some $\delta > 0$. Assume that P is searchable by some searcher,

[5] The subscripts for vertices, edges, and the points of a chord system are taken cyclically over $1, 2, \cdots, n$, so that $n + 1 = 1, n + 2 = 2$, etc.

[6] The symbols "p_i" and "e_j" are simply the labels of the points of the chord systems.

[7] We use the term "separable" for the vertices of P, and "separate(d)" for the points of a chord system.

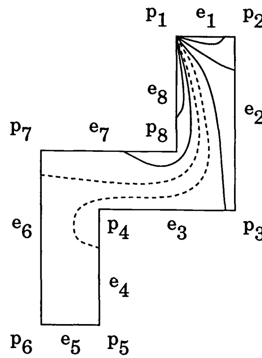


FIG. 13. *The chords incident on* p_1 *in* $\mathcal{C}_1(P)$ *and* $\mathcal{C}_2(P)$.

and let $p_{j_1}, p_{j_2}, \cdots, p_{j_K}$ be the order in which the vertices of P are hit during a search, where $j_1, j_2, \cdots, j_K \in \{1, 2, \cdots, n\}$ (ties are broken arbitrarily). Note that some vertices can appear more than once in the sequence $p_{j_1}, p_{j_2}, \cdots, p_{j_K}$. For each $1 \leq i \leq K$, we define Q_i to be the set of vertices that are clear at the moment p_{j_i} is hit, except that if more than one vertices are hit simultaneously, then we add these vertices to the set one at a time in the order they appear in the sequence $p_{j_1}, p_{j_2}, \cdots, p_{j_K}$. Formally, if $p_{j_i}, p_{j_{i+1}}, \cdots, p_{j_{i+\ell}}$, $\ell \geq 0$, are a maximal sequence of vertices that are hit simultaneously at time t, then we let Q be the set of clear vertices at t except $p_{j_i}, p_{j_{i+1}}, \cdots, p_{j_{i+\ell}}$, and define $Q_{i+k} = Q \cup \{p_{j_i}, p_{j_{i+1}}, \cdots, p_{j_{i+k}}\}$ for each $0 \leq k \leq \ell$. Then it is easy to verify the following:

(a) $Q_1 = \{p_{j_1}\}$;
(b) $Q_K = \{p_1, p_2, \cdots, p_n\}$ (since all vertices are clear at the end);
(c) For all $1 \leq i \leq K, p_{j_i} \in Q_i$;
(d) For all $2 \leq i \leq K, Q_i - \{p_{j_i}\} \subseteq Q_{i-1}$ (since a vertex not in Q_{i-1} can be included in Q_i only if it is p_{j_i});
(e) For all $1 \leq i \leq K$, any $x \in Q_i$ and $y \notin Q_i$ are separable (for the case of the 1-searcher) or 2-separable (for the case of the ∞-searcher) by p_{j_i} (since, otherwise, y is not illuminated and is thus contaminated by definition and there is a path between x and y not containing any illuminated point, and hence x must also be contaminated, a contradiction).

The sequence $p_{j_1}, p_{j_2}, \cdots, p_{j_K}$ is called a "feasible sequence" of a chord system in the next definition.

DEFINITION 7. Let $\mathcal{C} \subseteq A \times B$ be a chord system. For a subset $A_1 \subseteq A$, a sequence $\sigma : x_1, x_2, \cdots, x_K$ of points in A_1 is said to be a *feasible* (or *strongly feasible*) *sequence* of $\mathcal{C}$ with respect to A_1 if there exists a corresponding sequence $\Sigma : X_1, X_2, \cdots, X_K$ of subsets of A_1 satisfying the following conditions, (a)–(e) (or (a)–(d) and (e′)):

(a) $X_1 = \{x_1\}$;
(b) $X_K = A_1$;
(c) For all $1 \leq i \leq K, x_i \in X_i$;
(d) For all $2 \leq i \leq K, X_i - \{x_i\} \subseteq X_{i-1}$;
(e) For all $1 \leq i \leq K$, x_i separates X_i and $A_1 - X_i$;
(e′) For all $1 \leq i \leq K$, x_i strongly separates X_i and $A_1 - X_i$.

$\mathcal{C}$ is said to be *feasible* (or *strongly feasible*) if there exists a feasible (or strongly feasible) sequence of $\mathcal{C}$ with respect to A.

Example 7. Polygon P shown in Fig. 14 can be cleared by the 1-searcher using a schedule in which the vertices are hit in the order $p_1, p_2, p_3, p_8, p_5, p_9, p_4, p_5, p_7, p_6, p_7, p_8$. The positions of the 1-searcher and the beam of the flashlight when the vertices are hit are shown in the figure. Table 2 shows the vertices that are clear at the moment each vertex is hit. The reader can verify that the sequence given above is a feasible sequence of $C_1(P)$. The corresponding sequence of subsets of vertices is given in the second column of the table, namely, $\{p_1\}, \{p_1, p_2\}, \{p_1, p_2, p_3\}, \cdots$, etc.

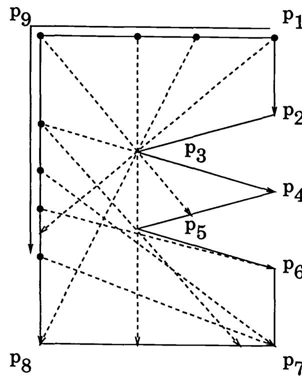


FIG. 14. *A search schedule of the* 1*-searcher.*

TABLE 2
Clear vertices at the moment each vertex is hit.

Vertex hit	Clear vertices at that moment
p_1	p_1
p_2	p_1, p_2
p_3	p_1, p_2, p_3
p_8	p_1, p_2, p_3, p_8
p_5	p_1, p_2, p_3, p_5
p_9	p_1, p_2, p_3, p_9
p_4	p_1, p_2, p_3, p_4, p_9
p_5	$p_1, p_2, p_3, p_4, p_5, p_9$
p_7	$p_1, p_2, p_3, p_4, p_5, p_7, p_9$
p_6	$p_1, p_2, p_3, p_4, p_5, p_6, p_9$
p_7	$p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_9$
p_8	$p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9$

The following two theorems are immediate from the definition and the discussion given above. We omit the proofs.

THEOREM 8. *Let P be a simple polygon. If P is* 1*-searchable, then $C_1(P)$ is feasible.*

THEOREM 9. *Let P be a simple polygon. If P is ∞-searchable, then $C_2(P)$ is feasible.*

A (strongly) feasible sequence σ of C is said to be *minimal* if no proper subsequence of it is a (strongly) feasible sequence.

PROPOSITION 2. *Let $\sigma : x_1, x_2, \cdots, x_K$ be a minimal strongly feasible sequence of $\mathcal{C} \subseteq A \times B$ with respect to A, and let $\Sigma : X_1, X_2, \cdots, X_K$ be a corresponding sequence of subsets of A satisfying conditions* (a)–(d) *and* (e′) *of Definition* 7. *For each $1 \leq i \leq K-1$, let $\overline{x_i y_i} \in \mathcal{C}$ be a chord separating X_i and $A - X_i$. Then for each $1 \leq i \leq K-1$, x_{i+1} is either the point not in X_i next to x_i or the point not in X_i next to y_i.*

Proof. Suppose that $x_{i+1} \in X_i$ for some $1 \leq i \leq K-1$, and let $k > i$ be the smallest index such that $x_k \notin X_i$. (Clearly $X_i \subset A$ by the minimality of σ.) Since $X_\ell \subseteq X_i$ for each $i+1 \leq \ell \leq k-1$ by condition (d), the sequence obtained from σ by removing subsequence $x_{i+1}, \cdots, x_{k-1}$ is a shorter strongly feasible sequence. Thus $x_i \notin X_i$. Also, if x_{i+1} is not one of the points not in X_i next to x_i or y_i, then $|A - X_i| \geq 3$, and hence by conditions (c), (d), and (e′) we have $X_{i+1} = \{x_{i+1}\}$. Therefore, $x_{i+1}, x_{i+2}, \cdots, x_K$ is a shorter strongly feasible sequence. This completes the proof. □

The strong feasibility of chord systems can be used to state sufficient conditions for the searchability of the given polygon.

THEOREM 10. *Let P be a simple polygon. If $\mathcal{C}_1(P)$ is strongly feasible, then P is* 1-*searchable.*

Proof. To save space, we only give an outline of the proof. Let $\sigma : p_{j_1}, p_{j_2}, \cdots, p_{j_K}$ be a minimal strongly feasible sequence of $\mathcal{C}_1(P)$ with respect to $A = \{p_1, p_2, \cdots, p_n\}$, and $\Sigma : Q_1, Q_2, \cdots, Q_K$ a corresponding sequence of subsets of A satisfying conditions (a)–(d) and (e′) of Definition 7. By condition (e′) for each $1 \leq i \leq K$ there exists a chord $\overline{p_{j_i} e_{h_i}} \in \mathcal{C}_1(P)$ that separates Q_i and $A - Q_i$. Since $\overline{p_{j_1} e_{j_1}} \in \mathcal{C}_1(P)$ and any chord incident to p_{j_1} separates $Q_1 = \{p_{j_1}\}$ and $A - Q_1$, we may assume that $e_{h_1} = e_{j_1}$. Similarly, we may assume that $e_{h_K} = e_{j_K}$. For each $1 \leq i \leq K$, let s_i be a point in edge e_{h_i} visible from $V(p_{j_i})$. Define Δ_i to be the situation in which (1) the 1-searcher is either at p_{j_i} or s_i, (2) the beam of the flashlight contains $\overline{p_{j_i} s_i}$, and (3) the vertices in Q_i are clear. Note that since $Q_K = \{p_1, \cdots, p_n\}$, P is clear in Δ_K. Thus we only need to show that situation Δ_K is reachable. We do this by induction on i. Since $Q_1 = \{p_{j_1}\}$, Δ_1 is realized if we place the 1-searcher at p_{j_1} and aim the flashlight at s_1. We show that Δ_i can be changed into Δ_{i+1} for any $1 \leq i \leq K-1$. At Δ_i, by Proposition 2 $p_{j_{i+1}}$ is either the vertex not in Q_i next to p_{j_i} or the vertex not in Q_i next to s_i. In the former case, we advance the endpoint of the beam at p_{j_i} to $p_{j_{i+1}}$ and move the other endpoint from s_i to s_{i+1}, in such a way that the vertices in Q_i which are also in Q_{i+1} remain clear. This is certainly possible if s_i and s_{i+1} are on the same edge of P. If s_i and s_{i+1} are not on the same edge, then by condition (d) it must be the case that the endpoint of the beam at s_i is moved backward to s_{i+1} over a portion of the boundary which is clear in Δ_i. (We observed a similar situation in Example 4 when the 2-searcher moved from $V(c)$ to $V(d)$.) See Fig. 15 for illustration. It is not hard to show that this backward movement can be done by a sequence of elementary actions so that the vertices in Q_i that are also in Q_{i+1} remain clear, regardless of whether the 1-searcher is at p_{j_i} or s_i in Δ_i. We leave the details to the reader. The latter case when $p_{j_{i+1}}$ is the point not in Q_i next to s_i is similar. □

THEOREM 11. *Let P be a simple polygon. If $\mathcal{C}_2(P)$ is strongly feasible, then P is* 2-*searchable.*

Proof. The proof is essentially similar to that of Theorem 10, except that we use the fact that the 2-searcher can illuminate simultaneously two points that are mutually 2-visible. We omit the details. □

$\mathcal{C}_1(P)$ and $\mathcal{C}_2(P)$ can be constructed in $O(n^2)$ time for an n-sided polygon P basically by constructing $V(x)$ and $V^2(x)$, respectively, in linear time from a triangulation of P for each vertex x of P [7], [21]. Given $\mathcal{C}_1(P)$ or $\mathcal{C}_2(P)$, we can find a minimal strongly feasible

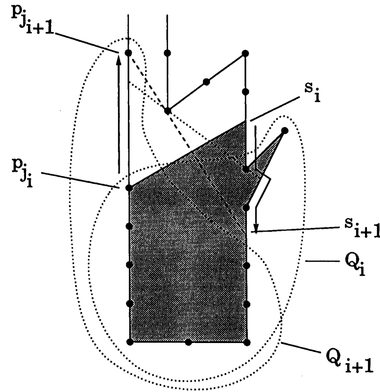


FIG. 15. *Movement of the endpoints of the beam.*

sequence $\sigma : p_{j_1}, p_{j_2}, \cdots$ of length $O(n^2)$ and a corresponding sequence $\Sigma : Q_1, Q_2, \cdots$ (or decide that no such sequences exist) in $O(n^2)$ time, as explained below. Note that the set Q_i corresponding to p_{j_i} consists of p_{j_i} and the points in $A = \{p_1, p_2, \cdots, p_n\}$ that are on one side of a chord incident on p_{j_i}. Thus for each $p \in A$, chord $\overline{pe}$ and a flag $s \in \{\mathbf{left}, \mathbf{right}\}$, we call the tuple $(p, \overline{pe}, s)$ a *configuration*. Configuration $(p, \overline{pe}, s)$ determines a set Q consisting of p and the points in A that are on the left (if $s = \mathbf{left}$) or right (if $s = \mathbf{right}$) of $\overline{pe}$ viewed from p. Then we can use $(p, \overline{pe}, s)$ to represent a pair of p and Q that appear correspondingly in σ and Σ. Since each point in A is an endpoint of $O(n)$ chords, there are $O(n^2)$ configurations. By Proposition 2 for each p_{j_i} and the corresponding Q_i there are at most two candidates for $p_{j_{i+1}}$, and for each candidate, clearly there exists a unique maximal subset of A that can be adopted as Q_{i+1}. This implies that there are at most two immediate successors for each configuration. Clearly, we can find a minimal strongly feasible sequence of length $O(n^2)$ (or know that no such sequence exists) by exploring the configuration space using breadth-first search repeatedly starting from various initial configurations, such as $(p_1, \overline{p_1e_1}, \mathbf{left})$ (which determines $\{p_1\}$), $(p_2, \overline{p_2e_2}, \mathbf{left})$ (which determines $\{p_2\}$), etc. Now we discuss the complexity of searching the configuration space. Given a configuration $(p_{j_i}, \overline{p_{j_i}e_{j_i}}, s_i)$ (representing a pair of p_{j_i} and Q_i), we can find the two candidates for $p_{j_{i+1}}$ in constant time. Note that continuing the search from a configuration $(p, \overline{pe}, \mathbf{left})$ is not necessary if another configuration $(p, \overline{pe'}, \mathbf{left})$ such that $\overline{pe'}$ is to the right of $\overline{pe}$ (viewed from p) has already been explored (since the latter determines a larger set than the former). Similarly, $(p, \overline{pe}, \mathbf{right})$ need not be explored any further if $(p, \overline{pe'}, \mathbf{right})$, such that $\overline{pe'}$ is to the left of $\overline{pe}$ (viewed from p), has already been explored. Thus for each $p \in A$, we maintain two chords $\overline{pe_{p,L}}$ and $\overline{pe_{p,R}}$, where $\overline{pe_{p,L}}$ (or $\overline{pe_{p,R}}$) is the rightmost (or leftmost) chord incident on p used in a reachable configuration with flag **left** (or **right**), which has already been found. If no such configuration has been found for the given flag value, then let $\overline{pe_{p,L}}$ (or $\overline{pe_{p,R}}$) be **nil**. Given a configuration $(p_{j_i}, \overline{p_{j_i}e_{j_i}}, s_i)$ and a candidate q for $p_{j_{i+1}}$, we first determine the flag s_{i+1} of the next configuration $(p_{j_{i+1}}, \overline{p_{j_{i+1}}e_{j_{i+1}}}, s_{i+1})$. This can be done in constant time without actually finding the chord $\overline{p_{j_{i+1}}e_{j_{i+1}}}$. Suppose that s_{i+1} is **left**. Then we test whether $\overline{p_{j_{i+1}}e_{j_{i+1}}}$ is to the right of $\overline{p_{j_i}e_{j_i}}$ in constant time (again, without finding $\overline{p_{j_{i+1}}e_{j_{i+1}}}$) by examining the relative positions of the endpoints of $\overline{qe_{q,L}}$ and $\overline{p_{j_i}e_{j_i}}$, and if so, then we obtain $\overline{p_{j_{i+1}}e_{j_{i+1}}}$ (as well as Q_{i+1}) by examining the chords incident on q sequentially (spending constant time per chord) from left to right (viewed from q) starting from the one immediately to the right of $\overline{qe_{q,L}}$ (or the leftmost chord if $\overline{qe_{q,L}}$ is

nil). The case when s_{i+1} is **right** is similar. Since this method ensures that each of the $O(n^2)$ chords is examined only a constant number of times during the entire process of searching, the complexity of searching for a minimal strongly feasible sequence is $O(n^2)$.

We conclude this section with the following technical proposition, which will be used in §5.

PROPOSITION 3. *If $\mathcal{C} \subseteq A \times B$ is feasible (or strongly feasible), then for any subset A_1 of A, there exists a feasible (or strongly feasible) sequence of $\mathcal{C}$ with respect to A_1.*

Proof. Let $\sigma : x_1, x_2, \cdots, x_K$ be a feasible (or strongly feasible) sequence of $\mathcal{C}$ with respect to A, and let $\Sigma : X_1, X_2, \cdots, X_K$ be a corresponding sequence of subsets of A satisfying conditions (a)–(e) (or (a)–(d) and (e′)) of Definition 7 with respect to A. For an arbitrary $A_1 \subseteq A$, let $\sigma' : x_{j_1}, x_{j_2}, \cdots, x_{j_\ell}$ be the sequence of points in A_1 obtained from σ by deleting the points not in A_1, and define the sequence $\Sigma' : Y_1, Y_2, \cdots, Y_\ell$ of subsets of A_1 by $Y_i = X_{j_i} \cap A_1$ for $1 \leq i \leq \ell$. It is a straightforward exercise to show that σ' and Σ' satisfy conditions (a)–(e) (or (a)–(d) and (e′)) of Definition 7 with respect to A_1. □

5. Additional necessary conditions. The goal of this section is to prove the following two theorems. Note that the non–∞-searchability of the polygon shown in Figs. 6 follows from Theorem 13.

THEOREM 12. *Let P be a simple polygon. If P is 1-searchable, then there exist vertices u and v of P such that P is weakly visible from $\pi(u, v)$.*

THEOREM 13. *Let P be a simple polygon. If P is ∞-searchable, then there exist vertices u and v of P such that P is weakly 2-visible from $\pi(u, v)$.*

A chord $\overline{x_1x_2} \in A \times A$ is called a *stabbing chord* of $\mathcal{C} \subseteq A \times B$ if x_1 and x_2 are separated by all $a \in A$. We use the following theorem to prove Theorems 12 and 13.

THEOREM 14. *Let $\mathcal{C}$ be a chord system. If $\mathcal{C}$ is feasible, then $\mathcal{C}$ has a stabbing chord.*

To prove Theorem 14, assume that there exists a feasible chord system having no stabbing chord, and let $\mathcal{C} \subseteq A \times B$ be such a chord system. We may assume that the size n of its base sets $A = \{a_1, \cdots, a_n\}$ and $B = \{b_1, \cdots, b_n\}$ is smallest among the base sets of all chord systems having the same property. Clearly $n \geq 2$. Let us first prove some properties of this $\mathcal{C}$ (Lemmas 1–5).

Since $\mathcal{C}$ has no stabbing chord, for any chord $\overline{x_1x_2} \in A \times A$, there exists a point $a \in A$ such that a does not separate x_1 and x_2. If there is *exactly one* such point a for $\overline{x_1x_2}$, then a is said to be *responsible* for $\overline{x_1x_2}$. For each $1 \leq i \leq n$, let Γ_i be the set of chords $c \in A \times A$ for which a_i is responsible.

LEMMA 1. *For any $1 \leq i \leq n$, $\Gamma_i \neq \emptyset$.*

Proof. Without loss of generality, assume that $\Gamma_n = \emptyset$. Construct a chord system $\mathcal{C}'$ having base set $A' = A - \{a_n\}$ of size $n-1$ from $\mathcal{C}$ by removing all chords incident to a_n and identifying b_{n-1} and b_n. Formally, $\mathcal{C}'$ is defined as follows:

1. For all i and j, $1 \leq i, j \leq n-1$, $\overline{a_ib_j} \in \mathcal{C}'$ if $\overline{a_ia_j} \in \mathcal{C}$;
2. For all i, $1 \leq i \leq n-1$, $\overline{a_ib_{n-1}} \in \mathcal{C}'$ if $\overline{a_ib_n} \in \mathcal{C}$.

Since $\mathcal{C}$ is feasible, there is a feasible sequence σ of $\mathcal{C}$ with respect to A' by Proposition 3. Then, by the construction of $\mathcal{C}'$, it is immediate that σ is also a feasible sequence of $\mathcal{C}'$ with respect to A'. Therefore, $\mathcal{C}'$ is feasible, and by the minimality of the base set of $\mathcal{C}$, $\mathcal{C}'$ must have a stabbing chord $\overline{x_1x_2} \in A' \times A'$. By the construction of $\mathcal{C}'$, each point $a_i \in A'$ separates x_1 and x_2 in $\mathcal{C}$. This implies that a_n must also separate x_1 and x_2 in $\mathcal{C}$, since, otherwise, a_n will be responsible for $\overline{x_1x_2}$, and hence $\overline{x_1x_2} \in \Gamma_n$. Therefore, $\overline{x_1x_2}$ is a stabbing chord of $\mathcal{C}$. This is a contradiction. □

LEMMA 2. *For any $1 \leq i, j \leq n$ such that $i \neq j$, $\Gamma_i \cap \Gamma_j = \emptyset$.*

Proof. Assume that $\overline{x_1x_2} \in \Gamma_i \cap \Gamma_j$ for some $i \neq j$. Then neither a_i nor a_j separates x_1 and x_2. Thus neither a_i nor a_j is responsible for $\overline{x_1x_2}$. This is a contradiction. □

Select a chord $f_i = \overline{s_it_i} \in \Gamma_i$ arbitrarily for each $1 \leq i \leq n$, and construct the set $F = \{f_i | 1 \leq i \leq n\}$.[8] By Lemma 2, we have $f_i \neq f_j$ for all $i \neq j$. Therefore, $|F| = n$.

LEMMA 3. *For each $a \in A$, the number of chords in F incident to a is exactly two.*

Proof. Since $|A| = |F| = n$, if there exists a point in A to which fewer than two chords in F are incident, then there must exist another point in A to which at least three chords in F are incident. Thus it suffices to show that the number of chords incident to each point in A is at most two. Assume that there are three distinct chords $\overline{ax_1}$, $\overline{ax_2}$, and $\overline{ax_3}$ in F which share $a \in A$ as an endpoint. Without loss of generality, assume that (1) points a, x_1, x_2, and x_3 appear in this order clockwise on the circumference of the unit circle, (2) $\overline{ax_2} \in \Gamma_i$ for some i, and (3) point a_i is in the arc containing x_1 and subtended by $\overline{ax_2}$. See Fig. 16. Since a_i does not separate a and x_2, it does not separate a and x_3 either. Since every chord in F has a unique point in A that does not separate its two endpoints, a_i must be the unique such point for $\overline{ax_3}$. Thus $\overline{ax_3} \in \Gamma_i$. Therefore, F contains two chords $\overline{ax_2}$ and $\overline{ax_3}$, both of which are in Γ_i. This contradicts the definition of F. □

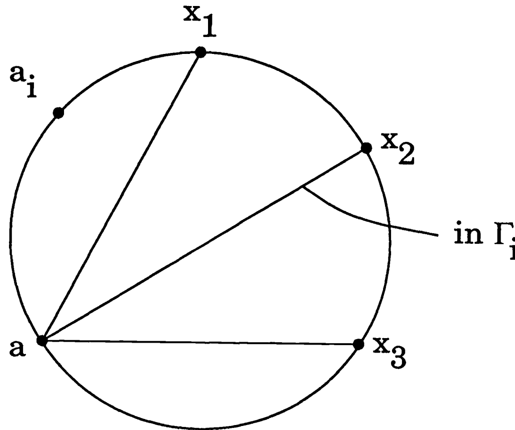


FIG. 16. *Illustration for the proof of Lemma* 3.

COROLLARY 1. *For any $1 \leq i \leq n$, $|\Gamma_i| = 1$.*

Proof. Assume that $|\Gamma_i| \geq 2$ for some i. By Lemma 3, the number of chords in F incident to each point in A is exactly two. Let F' be a set constructed from F by replacing f_i with another chord in Γ_i. Again by Lemma 3, the number of chords in F' incident to each point in A is exactly two. Obviously this is not possible. □

LEMMA 4. *For any $1 \leq i, j \leq n$ such that $i \neq j$, $f_i \cap f_j \neq \emptyset$.*[9]

Proof. Assume that F contains two chords $f_i = \overline{s_it_i} \in \Gamma_i$ and $f_j = \overline{s_jt_j} \in \Gamma_j$ such that $f_i \cap f_j = \emptyset$. Without loss of generality, assume that s_i, t_i, s_j and t_j appear in this order clockwise on the circumference of the unit circle. See Fig. 17. Since $\overline{s_it_i} \in \Gamma_i$ and $\overline{s_jt_j} \in \Gamma_j$, any $a_k \in A - \{a_i, a_j\}$ separates s_i and t_i, and also s_j and t_j. Thus any such a_k must separate s_i and s_j, and also t_i and t_j. If a_i separates s_i and s_j, then since $\overline{s_is_j}$ is not a stabbing chord, a_j must be responsible for $\overline{s_is_j}$, and hence $\overline{s_is_j} \in \Gamma_j$. Thus $|\Gamma_j| > 1$, which is a contradiction to Corollary 1. Therefore, a_i does not separate s_i and

[8] We will show later that each Γ_i is a singleton set, and hence F is in fact determined uniquely.

[9] Of course, such f_i and f_j may intersect at their endpoints.

s_j. By using a similar argument, we can show that a_i does not separate t_i and t_j either. This is impossible, however, since (1) s_j and t_j are separated by a_i and (2) any chord intersecting with $\overline{s_j t_j}$ must also intersect with either $\overline{s_i s_j}$ or $\overline{t_i t_j}$. □

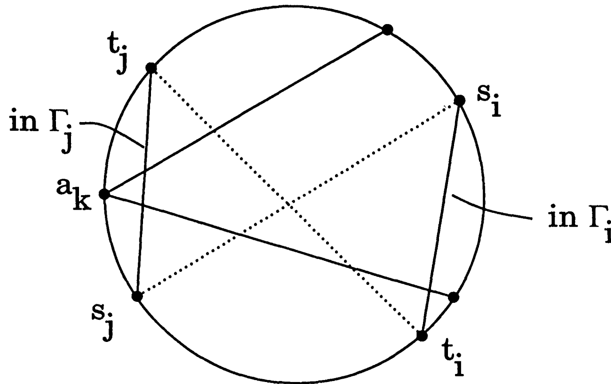


FIG. 17. *Illustration for the proof of Lemma* 4.

F is called a *star* if n is odd and $F = \{\overline{a_i a_{i+(n\pm1)/2}} \mid 1 \leq i \leq n\}$. In a star, each a_i is connected to the two points in A which are (almost) directly opposite to it. See Fig. 18.

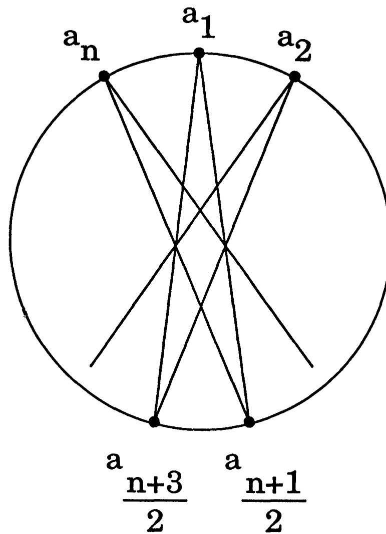


FIG. 18. *A star.*

LEMMA 5. *n is odd and F is a star.*

Proof. Let $f_i = \overline{s_i t_i}$ be any chord in F. Chord f_i divides the circumference of the unit circle into two arcs, each of which is subtended by f_i. Let ν_1 and ν_2, respectively, denote the numbers of points in $A - \{s_i, t_i\}$ lying in the two arcs, where $\nu_1 \leq \nu_2$. See Fig. 19. Since (1) $n = \nu_1 + \nu_2 + 2$, (2) f_i intersects with $n - 3$ chords in its interior, and (3) exactly two chords are incident to every point in A, we have

$$\left\lceil \frac{n-3}{2} \right\rceil \leq \nu_1 \leq \left\lfloor \frac{n-2}{2} \right\rfloor.$$

Assume that n is even. Then the inequalities given above yield $\nu_1 = \nu_2 = (n-2)/2$. Now let $f' = \overline{s_i t}$ be the chord in F which shares an endpoint s_i with f_i, and let ν_1' and ν_2', respectively, denote the numbers of points in $A - \{s_i, t\}$ lying in the two arcs subtended by f'. By using a similar argument we obtain $\nu_1' = \nu_2' = (n-2)/2$, but then this is certainly impossible. Thus n must be odd, $\nu_1 = (n-3)/2$, and $\nu_2 = (n-1)/2$. The fact that F is a star follows immediately. □

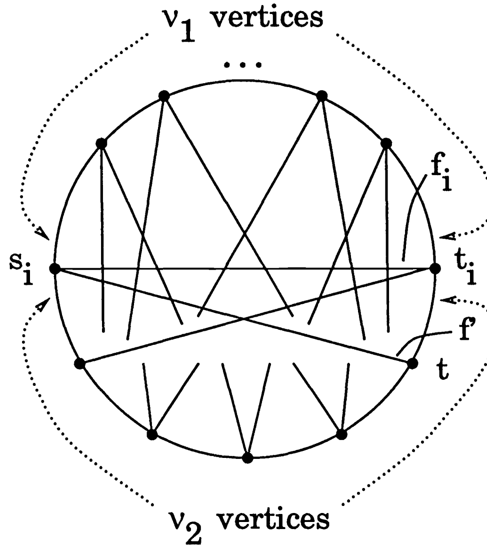


FIG. 19. *Illustration for the proof of Lemma* 5.

Proof of Theorem 14. We take C as above and derive a contradiction. Since C is feasible, let $\sigma = x_1, x_2, \cdots, x_K$ be a feasible sequence of C with respect to A. Let $\Sigma = X_1, X_2, \cdots, X_K$ be a sequence of subsets of A corresponding to σ satisfying conditions (a)–(e) of Definition 7. Let $\mu = (n+1)/2$. Since $X_i - \{x_i\} \subseteq X_{i-1}$ for all i, there is an i such that $|X_i| = \mu$. Let $x_i = a_k$ and $\Gamma_k = \{f_k\} = \{\overline{s_k t_k}\}$. Let A_1 denote the set of points in $A - \{s_k, t_k\}$ lying in the arc containing a_k and subtended by f_k, and let $A_2 = A - A_1$. See Fig. 20. Since $F = \{f_i \in \Gamma_i | 1 \le i \le n\}$ is a star by Lemma 5, $|A_1| \le (n-1)/2 < \mu$, and hence $X_i \cap A_2 \ne \emptyset$. But then we must have $A_2 \subseteq X_i$, since a_k does not separate any two points in A_2. (Note that $a_k \in A_1$ and a_k does not separate s_k and t_k.) Now let us estimate the value of $|X_i|$. Again by Lemma 5, we have $|A_2| \ge (n-3)/2 + 2 = \mu$. Also, $a_k \in X_i - A_2$. Thus, $|X_i| \ge \mu + 1$. This is a contradiction. □

Now we are ready to prove Theorems 12 and 13.

Proof of Theorem 12. The theorem follows from Theorems 8, 14, and the observation that if $\overline{a_i a_j}$ is a stabbing chord of $C_1(P)$, then every vertex of P (and hence P itself) is weakly visible from $\pi(p_i, p_j)$. □

Proof of Theorem 13. The theorem follows from Theorems 9, 14, and the observation that if $\overline{a_i a_j}$ is a stabbing chord of $C_2(P)$, then every vertex of P (and hence P itself) is weakly 2-visible from $\pi(p_i, p_j)$. □

6. Hedgehogs. In this section we introduce a class of polygons called hedgehogs and show that any ∞-searchable hedgehog is 2-searchable. The complexity of generating a search schedule for a hedgehog is also discussed. Again, chord systems, which are constructed slightly differently from those in §4, are the main tool in this section.

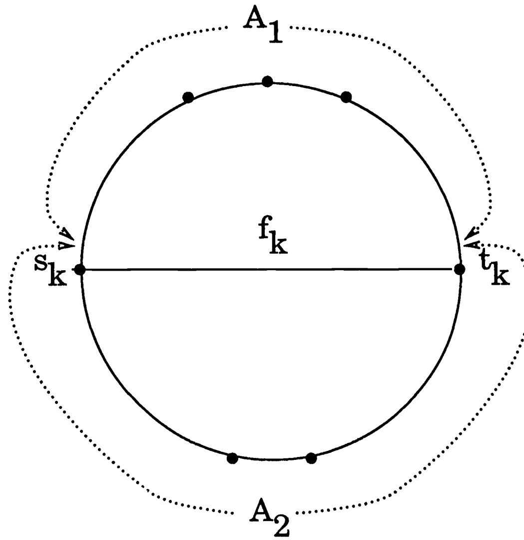


FIG. 20. *Illustration for the proof of Theorem* 14.

A *hedgehog* is a simple polygon consisting of a convex body and a set of narrow hooked pins (corridors). Examples of hedgehogs are shown in Figs. 3, 5, and 6. More specifically, a hedgehog is constructed from a convex polygon by replacing some of its edges by a polygonal chain such as the one shown in Fig. 21 consisting of four segments representing a hooked pin under a restriction stated below.

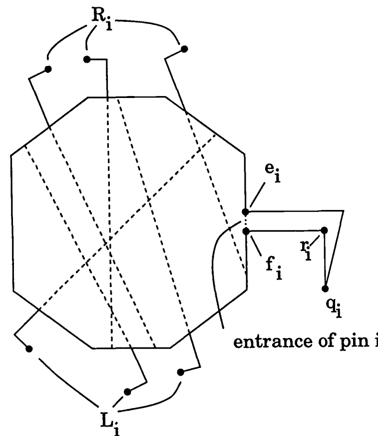


FIG. 21. q_i, r_i, e_i, f_i, L_i, *and* R_i *of pin* i.

Let us introduce the following symbols (see Fig. 21). Let P be an n-sided hedgehog having m hooked pins. Let $p_1, p_2, \cdots, p_n$ be the vertices of P taken clockwise. We denote by $q_1, q_2, \cdots, q_m$ the tips of the hooked pins, where for some $1 \leq j_1 < j_2 < \cdots < j_m \leq n$, $q_i = p_{j_i}$ for $1 \leq i \leq m$. For convenience, the hooked pin containing q_i is called pin i. For $1 \leq i \leq m$, q_i and r_i denote the tip of pin i and the reflex vertex adjacent to q_i, respectively. Also, we denote by e_i and f_i the points where the edges forming pin i meet the convex body, and call the line segment $\overline{e_i f_i}$ the *entrance* of pin i. We require that the

pins are sufficiently narrow and appropriately oriented so that no point in the entrance of a pin is 2-visible from the tip of any other pin. (This implies that, when we construct a hedgehog from a convex polygon, we should not replace two adjacent edges by pins.) Finally, for each $1 \leq i \leq m$, L_i (or R_i) denotes the set of the tips of the hooked pins that are to the left (or right) of $V^2(q_i)$ when viewed from r_i. (Neither L_i nor R_i contains q_i.)

THEOREM 15. *Let P be a hedgehog having m hooked pins. Then P is 1-searchable if and only if $m \leq 2$.*

Proof. It is obvious that P is 1-searchable if $m \leq 2$. If $m \geq 3$, then the tips of any three hooked pins are mutually nonseparable, and hence P is not 1-searchable by Theorem 3. □

In the following we characterize 2- and ∞-searchable hedgehogs having three or more hooked pins. Let P be a hedgehog having $m \geq 3$ pins. Define a chord system $\mathcal{C}_H(P)$ of P with base sets $A = \{q_1, q_2, \cdots, q_m\}$ and $B = \{b_1, b_2, \cdots, b_m\}$ by $\overline{q_i b_j} \in \mathcal{C}_H(P)$ if and only if q_j and q_{j+1} are 2-separable by q_i in P. (In a sense, point b_j represents the portion of the boundary of P between the entrances of pins j and $j+1$.) Since no point in the entrance of a pin is 2-visible from the tip of any other pin, each $q_i \in A$ is an endpoint of at most three chords in $\mathcal{C}_H(P)$, namely, $\overline{q_i b_{i-1}}$, $\overline{q_i b_i}$ and possibly $\overline{q_j b_j}$ for some $j \neq i-1, i$.

LEMMA 6. *Let P be a hedgehog. If P is ∞-searchable, then $\mathcal{C}_H(P)$ is strongly feasible.*

Proof. Assume that P is ∞-searchable, and let $q_{j_1}, q_{j_2}, \cdots, q_{j_K}$ be the order in which the tips of the hooked pins of P are hit during a search by the ∞-searcher, where $j_1, j_2, \cdots, j_K \in \{1, 2, \cdots, m\}$. For each $1 \leq i \leq K$ let Q_i be the set of clear tips at the moment q_{j_i} is hit. Clearly $q_{j_1}, q_{j_2}, \cdots, q_{j_K}$ and $Q_1, Q_2, \cdots, Q_K$ satisfy conditions (a)–(d) of Definition 7. Since no two tips in L_{j_i} (and no two tips in R_{j_i}) are separable by the position of the ∞-searcher when q_{j_i} is hit, Q_i is either $\{q_{j_i}\}$, $\{q_{j_i}\} \cup L_{j_i}$, $\{q_{j_i}\} \cup R_{j_i}$, or $\{q_1, q_2, \cdots, q_m\}$. Thus Q_i and $A - Q_i$ are strongly separated by q_{j_i} in $\mathcal{C}_H(P)$ for all $1 \leq i \leq K$. Therefore, $\mathcal{C}_H(P)$ is strongly feasible. □

Example 8. For the hedgehog shown in Fig. 3 examined in Example 3, the sequence a, b, g, c, d, a, e, f, c, g, b, a given in the first column of Table 1 is a minimal strongly feasible sequence of its chord system. The corresponding sequence of subsets of tip vertices is given in the second column, namely, $\{a\}$, $\{a, b\}$, $\{a, b, g\}, \cdots$, etc.

LEMMA 7. *Let P be a hedgehog. If $\mathcal{C}_H(P)$ is strongly feasible, then P is 2-searchable.*

Proof. We only give an outline of the proof. Let $\sigma : q_{j_1}, \cdots, q_{j_K}$ be a minimal strongly feasible sequence of $\mathcal{C}_H(P)$ with respect to $A = \{q_1, q_2, \cdots, q_m\}$. Let $\Sigma : Q_1, \cdots, Q_K$ be a corresponding sequence of subsets of A. The 2-searcher can visit $V(q_{j_1}), V(q_{j_2}), \cdots$, $V(q_{j_K})$ in this order in such a way that the tips in Q_i are clear when he is in $V(q_{j_i})$. Initially, the 2-searcher is in $V(q_{j_1})$ aiming both flashlights at q_{j_1}. (Note that $Q_1 = \{q_{j_1}\}$.) Suppose that at present the 2-searcher is in $V(q_{j_i})$ for some $1 \leq i \leq K-1$ and the tips in Q_i are clear. By Proposition 2, the tip $q_{j_{i+1}}$ that he must clear next is one of the contaminated tips right next to the clear tips. Then it is easy to show that the 2-searcher can move to $V(q_{j_{i+1}})$ advancing one flashlight to $q_{j_{i+1}}$ and possibly moving the other backward, and clear $q_{j_{i+1}}$ without recontaminating any tips in Q_i which are also in Q_{i+1}. (A similar argument is used in the proof of Theorem 10.) When he reaches $V(q_{j_K})$ all tips are clear (note that $Q_K = \{q_1, q_2, \cdots, q_m\}$), and again it is easy to show that he can clear the remaining contaminated points (if any) without recontaminating any tip. We leave the details to the reader. □

By Lemmas 6 and 7 and the fact that any 2-searchable polygon is ∞-searchable (Proposition 1), we obtain the following theorem.

THEOREM 16. *Let P be a hedgehog. The following three statements are equivalent:*

1. *P is ∞-searchable;*
2. *P is 2-searchable;*
3. *$\mathcal{C}_H(P)$ is strongly feasible.*

Finally, we discuss the time complexity of computing a search schedule for hedgehogs.

LEMMA 8. *Let $\mathcal{C}_H(P)$ be the chord system of a hedgehog P having m hooked pins. Given a suitable representation of $\mathcal{C}_H(P)$, whether $\mathcal{C}_H(P)$ is strongly feasible can be tested, and if so, a minimal strongly feasible sequence of length $O(m)$ of $\mathcal{C}_H(P)$ with respect to A can be obtained in $O(m)$ time.*

Proof. The argument is similar to that given in §4 (following Theorem 11) regarding $\mathcal{C}_1(P)$ and $\mathcal{C}_2(P)$. The difference is that each point in A is an endpoint of at most three chords in $\mathcal{C}_H(P)$, instead of $O(n)$ chords. Therefore (1) there are only $O(m)$ possible pairs of q_{j_i} and Q_i, and (2) the binary search we used for finding Q_{i+1} for the given candidate for $q_{j_{i+1}}$ is no longer necessary. We omit the details to save space. □

THEOREM 17. *Let P be an n-sided hedgehog having m hooked pins. Whether P is 2-searchable can be tested, and if so, a search schedule of the 2-searcher consisting of $O(m)$ elementary actions for clearing P can be generated in $O(n + m \log n)$ time.*

Proof. Given the vertices $p_1, p_2, \cdots, p_n$ of P, we first find the tips $q_1, q_2, \cdots, q_m$. The reader can verify that this can be done in $O(n)$ time. (For example, we look for a maximal run of reflex vertices in $p_1, \cdots, p_n$. Clearly the length of such a run is at most four. If there are four consecutive reflex vertices $p_i, \cdots, p_{i+3}$, then both p_{i-1} and p_{i+4} are tips. If p_i, p_{i+1} is a maximal sequence of reflex vertices, then either p_{i-1} or p_{i+2} is a tip, and p_{i+2} is a tip if and only if p_{i+3} is nonreflex and p_{i+4} is not visible from p_{i+2}. Whether or not p_{i+4} is visible from p_{i+2} can be tested in $O(n)$ time. The case when there is a maximal run of three reflex vertices is similar, and we can find a tip in $O(n)$ time. Once a tip is found, all other tips can be found in $O(n)$ time.) Then a representation of $\mathcal{C}_H(P)$ suitable for the purpose of the method given in the proof of Lemma 8 can be obtained by testing the visibility from r_i (the reflex vertex adjacent to q_i) for each $1 \le i \le m$. Since the body of P is convex, this can be done in $O(\log n)$ time for each r_i by finding, using binary search, the intersection of the convex body of P and the line containing r_i and one of e_i and f_i. The total time so far is $O(n + m \log n)$. When this is done, we test whether $\mathcal{C}_H(P)$ is strongly feasible, and if so, we find a minimal strongly feasible sequence σ of length $O(m)$ in $O(m)$ time, using the method given in the proof of Lemma 8. Finally, from σ we generate a search schedule for the 2-searcher described in the proof of Lemma 7. Since P is a hedgehog having a special structure, the 2-searcher can move from $V(q_{j_i})$ to $V(q_{j_{i+1}})$ and clear $q_{j_{i+1}}$ (without contaminating the tips in Q_i which are also clear in Q_{i+1}) by a sequence of $O(1)$ elementary actions, and such a sequence can be generated in $O(\log n)$ time using bullet shooting (again, bullet shooting reduces to binary search). The length of the entire schedule generated is certainly $O(m)$ and the time needed to generate it is $O(m \log n)$. The overall time, therefore, is $O(n + m \log n)$. □

Finally, we briefly discuss a variant of hedgehogs. Given a hedgehog P, construct P' from P by stretching all hooked pins. (P' can be viewed as a hedgehog with straight pins.) By Theorem 6 such P' is always 2-searchable. It turns out that the 1-searchability of P' can be determined by constructing a chord system (as we did for P) using the visibility (instead of the 2-visibility) from the tips of the pins, and then testing whether it is (strongly) feasible. Since, of course, the chord system constructed above for P' is identical to $\mathcal{C}_H(P)$, we obtain the following theorem. The proof is omitted.

THEOREM 18. *Let P be a hedgehog. Let P' be a simple polygon constructed from P by replacing the hooked pins by straight pins. Then P is 2-searchable if and only if P' is 1-searchable.*

7. Concluding remarks. We have posed the polygon search problem and presented some necessary conditions and sufficient conditions for a polygon to be searchable by various searchers. The related complexity issues have also been discussed. We have also shown that the 2-searcher is as capable as the ∞-searcher if the given polygon is a hedgehog. Chord systems have played a major role in many of the discussions. An interesting open problem is to prove or disprove the conjecture that the 2-searcher can search any polygon searchable by the ∞-searcher. This problem, as well as the problem of determining the complexity of deciding whether a given polygon is searchable by a given searcher in the general case, is suggested for future research.

Acknowledgments. We would like to thank the anonymous referees for their helpful comments on an earlier version of this paper.

REFERENCES

[1] D. AVIS AND G. T. TOUSSAINT, *An optimal algorithm for determining the visibility of a polygon from an edge*, IEEE Trans. Comput., C-30 (1981), pp. 910–914.

[2] B. CHAZELLE, *Triangulating a simple polygon in linear time*, Proc. of the 31st Symposium on Foundations of Computer Science, St. Louis, MO, October, 1990, pp. 220–230.

[3] W.-P. CHIN AND S. NTAFOS, *Optimum watchman routes*, Proc. of the 2nd Annual Symposium on Computational Geometry, Yorktown Heights, NY, June, 1986, pp. 24–33.

[4] ———, *Shortest watchman routes in simple polygons*, Tech. Report, Computer Science Dept., University of Texas, Dallas, TX, 1987.

[5] V. CHVÁTAL, *A combinatorial theorem in plane geometry*, J. Combin. Theory Ser. B, 18 (1975), pp. 39–41.

[6] H. EDELSBRUNNER, H. A. MAUER, F. P. PREPARATA, A. ROSENBERG, E. WELZL, AND D. WOOD, *Stabbing line segments*, BIT, 22 (1982), pp. 274–281.

[7] L. GUIBAS, J. HERSHBERGER, D. LEVEN, M. SHARIR, AND R. TARJAN, *Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons*, Algorithmica, 2 (1987), pp. 209–233.

[8] J. KAHN, M. KLAWE, AND D. KLEITMAN, *Traditional galleries require fewer watchmen*, SIAM J. Algebraic Discrete Meth., 4 (1983), pp. 194–206.

[9] Y. KE, *Polygon visibility algorithms for weak visibility and link distance problems*, Ph.D. dissertation, Department of Computer Science, Johns Hopkins University, Baltimore, MD, 1989.

[10] A. S. LAPAUGH, *Recontamination does not help to search a graph*, TR-335, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1985.

[11] D. T. LEE AND A. K. LIN, *Computational complexity of Art Gallery problems*, IEEE Trans. Inform. Theory, IT-32 (1986), pp. 276–282.

[12] D. T. LEE AND F. P. PREPARATA, *An optimal algorithm for finding the kernel of a polygon*, J. ACM, 26 (1979), pp. 415–421.

[13] ———, *Euclidean shortest paths in the presence of rectilinear barriers*, Networks, 14 (1984), pp. 393–410.

[14] N. MEGIDDO, S. L. HAKIMI, M. R. GAREY, D. S. JOHNSON, AND C. H. PAPADIMITRIOU, *The complexity of searching a graph*, J. ACM, 35 (1988), pp. 18–44.

[15] J. O'ROURKE, *An alternate proof of the rectilinear Art Gallery theorem*, J. Geom., 21 (1983), pp. 118–130.

[16] ———, *Art Gallery Theorems and Algorithms*, Oxford University Press, New York, 1987.

[17] T. D. PARSONS, *Pursuit-evasion in a graph*, in Theory and Applications of Graphs, Y. Alavi and D. Lick, eds., Lecture Notes in Mathematics 642, Springer-Verlag, Berlin, 1976, pp. 426–441.

[18] J.-R. SACK AND S. SURI, *An optimal algorithm for detecting weak visibility of a polygon*, Proc. of the 5th Symposium on Theoretical Aspects of Computer Science, Springer-Verlag, 1988, pp. 312–321.

[19] T. SHERMER, *Recent results in art galleries*, TR90-10, Computing Science, Simon Fraser University, 1990.

[20] K. SUGIHARA, I. SUZUKI, AND M. YAMASHITA, *The searchlight scheduling problem*, SIAM J. Comput., 19 (1990), pp. 1024–1040.

[21] S. SURI, *Minimum link paths in polygons and related problems*, Ph.D. dissertation, Department of Computer Science, Johns Hopkins University, Baltimore, MD, 1987.

SIAM J. COMPUT.
Vol. 21, No. 5, pp. 889–895, October 1992
© 1992 Society for Industrial and Applied Mathematics
005

DETERMINISM VS. NONDETERMINISM IN MULTIPARTY COMMUNICATION COMPLEXITY*

DANNY DOLEV† AND TOMÁS FEDER‡

Abstract. A given Boolean function has its input distributed among many parties. The aim is to determine which parties to talk to and what information to exchange in order to evaluate the function while minimizing the total communication. This paper shows that it is possible to evaluate the Boolean function deterministically with only a polynomial increase in communication and number of parties accessed with respect to the information lower bound given by the nondeterministic communication complexity of the function.

Key words. communication complexity, multiparty communication

1. Introduction. Our model of multiparty communication complexity is motivated by two basic earlier models. The two-party communication model assumes that each of two processors has a part of the input, and the aim is to compute a function on the input minimizing the amount of communication. In the decision tree model, the input is distributed among many memory locations, and the aim is to compute a function on the input while minimizing the number of memory locations examined. Our multiparty communication model extends these two basic models by assuming that the input is distributed among many processors; here the goal is to minimize both communication and number of processors accessed.

Two-party communication has been extensively studied. The main issues studied were the relative power of determinism, nondeterminism, and randomization. Yao [19] introduced the tool of minimum fooling set (or crossing sequence) as a measure for the amount of information that needs to be exchanged for a given input partitioned among the two parties. The same technique was widely used in [2], [7], [9], [11], [13].

The decision tree model has been studied in several contexts [3], [10], [12], [15], [16], [17]. An area that inspired research in this direction is the study of graph properties (see [14], for example). The main focus in these studies is how to minimize the fraction of the input that must be examined in order to verify a given property. Here again we are interested in the relative power of determinism, nondeterminism, and randomization. The basic issue is how to decide what input locations to examine. Similar reduction ideas appear in the proof of Theorem 1 in [1].

In the multiparty communication model, when a large amount of information is distributed among a large number of processors, it is crucial to decide both which processors to communicate with and what information to exchange. We can neither talk to all parties as in the two-party model, nor obtain all the information known to each party as in the decision tree model. A natural measure for the least amount of information required is the information that a nondeterministic algorithm needs to exchange in order to decide the value of the function. In this paper we show that when computing a Boolean function, this information can be obtained deterministically with limited overhead. More precisely, we prove that the deterministic and the nondeterministic communication complexity of multiparty Boolean function evaluation are polynomially related.

Tight bounds relate the deterministic and the nondeterministic communication complexity in the two-party model. Let C_1 be the nondeterministic communication complex-

* Received by the editors December 20, 1989; accepted for publication (in revised form) August 5, 1991.
† Computer Science Department, Hebrew University, Jerusalem 91904, Israel, and IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120.
‡ IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120.

ity of the language defined by a Boolean function $f(x_1, x_2)$, and C_0 that of its complement. Aho, Ullman, and Yannakakis [2] showed that the deterministic complexity of f is at most $O(C_0C_1)$; Halstenberg and Reischuk [9] improved this bound to $C_0C_1(1+o(1))$. A matching lower bound was obtained by Halstenberg and Reischuk [9], improving an earlier result of Mehlhorn and Schmidt [11]. Fürer [8] obtained similar lower bounds for the randomized case. Further restrictions on the communication exchange, such as bounding the number of rounds, have been studied by Papadimitriou and Sipser [13]; Duris, Galil, and Schnitger [7]; and others.

Quadratic bounds relating deterministic and nondeterministic complexities have also been obtained for decision trees. Let k_1 and k_0 be the nondeterministic complexity (the number of memory locations examined) of a Boolean function of n variables $f(x_1, \cdots, x_n)$ and its complement. Blum and Impagliazzo [3], Hartmanis and Hemachandra [10], and Tardos [17] independently showed that the deterministic decision tree complexity of f is at most k_0k_1. Related results for randomized decision trees can be found in Saks and Wigderson [16] and Nisan [12].

Our work was motivated by the striking similarity of the results in these two models, which give quadratic C_0C_1 and k_0k_1 bounds, respectively. The methods used to obtain the bounds in these two models, however, are very different. Since in distributed computing, the natural model is one that combines both, we should wonder whether a similar relation holds for multiparty communication. Our result gives a bound on the order of ${k_0}^2k_1$ for the number of parties accessed with C_0C_1 bits exchanged with each one, up to logarithmic factors, where k_1 and C_1 are the number of parties accessed and the total number of bits exchanged in a nondeterministic algorithm for f, and k_0 and C_0 are the analogous parameters for the complementary function $1 - f$. This bound essentially matches the communication bound for the two-party case while only increasing the bound on the number of parties accessed by a factor of k_0 with respect to the decision tree case. It improves the bound $(k_0C_0)^2(k_1C_1)$ on the total communication from an earlier version of this paper by a factor of C_0 [6].

Communication complexity in distributed computing has mainly focused on the number of messages or bits required to compute a specific function in a system. The complexity usually arises from either symmetry breaking or asynchronous behavior. The only study that is somewhat close to ours was done by Tiwari [18]. Tiwari mainly studies a chain of processors computing a function $f(x_1, x_2)$, where the inputs are at both ends of the chain. The difficulties in this model are knowing what information to distribute (as in the two-party model) and how that information should be propagated along the chain. In this model the added complexity of deciding what processors to query does not arise.

In order to concentrate on the combined complexity of deciding what processors to query and what information to exchange with them, we assume the following model. The input is distributed among n parties, and a single *coordinator* can communicate directly with each one of them. We can easily show that allowing direct communication among the parties will not significantly affect the bounds that we obtain.

In [5] a different communication complexity model was defined. In that model each party has all the inputs but one, and all parties communicate through a shared "blackboard." This model was also used in [4]. Our results do not apply to this model because the inputs that individual parties hold are not independent.

2. Definitions. Suppose that a coordinator wishes to evaluate a Boolean-valued function $f(x_1, \cdots, x_n)$, where each x_i is chosen from an arbitrary set Γ_i. The input vector $x = (x_1, \cdots, x_n)$ is distributed among n parties, with x_i known to party i.

We shall define nondeterministic algorithms in terms of communication behavior. A nondeterministic algorithm $\mathcal{N}_1$ that accepts the language defined by f (the set of input vectors that map to 1 under f) is a tuple $(S_1, \cdots, S_n, A_1, \cdots, A_n, V^1)$. The components of such a tuple are as follows. Each S_i is a set of nonempty binary sequences that represents the possible communication exchanges between the coordinator and party i. The binary sequences in S_i are *self-delimiting*, i.e., no one is a prefix of another. (This makes it possible to uniquely determine the end of the sequence.) Each A_i is a function that maps each sequence $s_i \in S_i \cup \{\epsilon\}$ to a nonempty subset $A_i(s_i)$ of Γ_i; this subset represents the possible inputs at party i for which s_i is a valid communication from the point of view of party i. Here ϵ is the empty sequence and represents the case where no communication occurs between the coordinator and party i; we thus require that $A_i(\epsilon) = \Gamma_i$. A *communication vector* $s = (s_1, \cdots, s_n)$ with $s_i \in S_i \cup \{\epsilon\}$ *covers* an input vector $x = (x_1, \cdots, x_n)$ at party i if $x_i \in A_i(s_i)$. Furthermore, x is *consistent* with s if s covers x at each party i. We say that party i is *accessed* by s if s_i is nonempty. The communication vector s is a 1-*certificate* if $f(x) = 1$ for all x consistent with s. The last component V^1 is a set of 1-certificates such that each input vector x with $f(x) = 1$ is consistent with some $s \in V^1$, and represents the communication vectors that are accepted by the coordinator.

We characterize the communication complexity of $\mathcal{N}_1$ with two parameters. The first parameter C_1 is the maximum over all 1-certificates $s \in V^1$ of $\sum_i \text{length}(s_i)$; thus C_1 is the maximum number of bits exchanged when $\mathcal{N}_1$ accepts. The second parameter k_1 is the maximum over all 1-certificates $s \in V^1$ of the number of parties accessed by s; thus k_1 is the maximum number of parties accessed when $\mathcal{N}_1$ accepts. We also assume the existence of a nondeterministic algorithm $\mathcal{N}_0$ that accepts the language defined by the complementary function $1 - f$, and define 0-certificates, V^0, C_0, k_0, and the appropriate terminology analogously.

We say that a 1-certificate s and a 0-certificate t are *incompatible* at party i if $A_i(s_i) \cap A_i(t_i) = \emptyset$. Notice that every 0-certificate must be incompatible with every 1-certificate somewhere because otherwise we could construct an input vector on which f takes both values 0 and 1.

3. A deterministic algorithm. The algorithm of Blum and Impagliazzo [3] for the decision tree model works by repeatedly "exposing" the parties accessed by given 1-certificates in turn; each 1-certificate chosen for this purpose is required to cover the input at parties exposed earlier by previous 1-certificates. By incompatibility, if t is a 0-certificate that covers the input at the parties already exposed, then the next 1-certificate s chosen must expose a new party accessed by both s and t. Thus by the time k_0 1-certificates have been chosen, any 0-certificate consistent with the input has been completely exposed, and the value of f can be verified directly. The total number of parties exposed is at most $k_0 k_1$.

A straightforward adaptation of this approach does not work in our model. The reason is that it is too expensive to obtain all the information stored at each party exposed. To overcome this difficulty, we choose a set of parties to expose. Each party exposed evaluates with respect to its input, those 1-certificates that were not yet discarded. It communicates enough information, via a 0-certificate that covers its input, to discard a fraction of the possible 1-certificates left. To keep the amount of information "wasted" bounded, it does not communicate when this implies discarding only a very small fraction. Only when no exposed party has a valuable contribution does the coordinator use the remaining 1-certificates to choose more parties to expose. Every time the set of exposed parties increases, the number of exposed accessed parties for each 0-certificate

consistent with the input increases as well, as in the decision tree algorithm. By the $k_0 + 1$th time the value of the function is determined.

The following lemma will be important in bounding the amount of communication required by the algorithm.

LEMMA 3.1. *If the Boolean function f has nondeterministic complexity bounded by C_0, k_0, C_1, k_1, then there exists a nondeterministic algorithm for f for which the set of 1-certificates satisfies $|V^1| \leq 2^{C_1} {k_0}^{k_1}$.*

Proof. The number of 1-certificates in V^1 is at most $2^{C_1} n^{k_1}$, since each certificate s is described by the C_1 bits communicated and the k_1 out of n parties accessed. We show that the dependency on n can be eliminated by replacing n with the potentially smaller k_0. Consider the list of all 0-certificates in V^0 in some canonical order (say, the lexicographic order). Choose a 1-certificate s in V^1, and produce the following description. For each 0-certificate t in the canonical list in turn, find a party i at which s is incompatible with t, indicate which of the k_0 parties accessed in t is party i, and then give the sequence s_i that characterizes the communication with party i. Delete then from the list all 0-certificates that are incompatible with s at party i. When the end of the list is reached, the description contains at most C_1 communication bits and k_1 parties described by a number in the range $1, \cdots, k_0$, for a total of $2^{C_1} {k_0}^{k_1}$ possible descriptions. The communication vector s' indicated by this description may be smaller than the 1-certificate s, since only a fraction of the parties accessed by s is listed in the description. On the other hand, by construction, each 0-certificate t in V^0 is incompatible with s', and so s' is indeed a 1-certificate. The certificate s' can, in fact, be recovered from the description by traversing the canonical list and identifying the appropriate parties. Thus the number of 1-certificates s' obtained by this construction is indeed bounded by $2^{C_1} {k_0}^{k_1}$. □

We now describe a deterministic algorithm for a Boolean function f. In this algorithm, all communication is initiated by the coordinator, who sends messages to various parties in turn and receives a response from each of them. Just like in the conventional two-party model, each party knows the protocol in advance and uses its own local memory during the execution. When the algorithm terminates, the coordinator must hold the value of f.

THEOREM 3.1. *There is a deterministic algorithm for f that communicates with a total of $2k_0^2 k_1$ parties and exchanges $2(C_1 + \lceil k_1 \log k_0 \rceil + 1)(C_0 + k_0(\lceil \log(2k_0^2 k_1) \rceil + 2))$ bits with each.*

Proof. The deterministic algorithm for computing $f(x_1, \cdots, x_n)$ maintains two sets: a set of chosen parties, the *exposed* parties, and a set of candidate 1-certificates from V^1, the *current* 1-certificates. The algorithm runs in $k_0 + 1$ phases and satisfies the following basic properties.

(i) All communication during a phase occurs only between the coordinator and exposed parties.

(ii) All information sent by an exposed party to the coordinator is shared with all of the exposed parties, so that every exposed party can deduce the set of current 1-certificates.

(iii) New parties are exposed only at the end of a phase.

(iv) If the value of the function is 0 then at the beginning of phase j, each 0-certificate consistent with the input accesses at least j exposed parties.

Each phase discards some 1-certificates that are not consistent with the input vector $x = (x_1, \cdots, x_n)$, by communicating 0-certificates that cover the input at some exposed party to all other exposed parties. If it is no longer possible to discard a large fraction of the 1-certificates in this way with a reduced amount of communication, then we shall

show that the following property must hold: each 0-certificate t in V^0 consistent with the input must be incompatible with at least half of the current 1-certificates at *nonexposed* parties. This property implies that such a t must be incompatible with at least a fraction $1/(2k_0)$ of the current 1-certificates at *some* nonexposed party (since at most k_0 parties are accessed by t). We then expose all nonexposed parties accessed by a fraction $1/(2k_0)$ of the current 1-certificates; this exposes, in particular, at least one more party accessed by t, for each 0-certificate t in V^0 consistent with the input. If the set of current 1-certificates is still nonempty, we proceed to the next phase.

By the time the (k_0+1)th phase is executed, if all 1-certificates have been discarded, then the value of f is 0; otherwise k_0+1 parties are accessed by every 0-certificate t in V^0 consistent with the input; this is impossible unless no such certificate exists, in which case the value of f is 1.

Each phase thus consists of two steps: The first step reduces the number of current 1-certificates. The second step increases the number of exposed parties (and implicitly the number of exposed parties for 0-certificates consistent with the input). The two steps are given below in full detail. Note that, at the beginning of the first phase, step (1) can be skipped since no exposed parties have been chosen yet, and that step (2) need not be executed in the (k_0+1)th and last phase because by that time the value of f is already determined by whether the set of current 1-certificates is empty or not.

(1) Each exposed party i, in turn, looks for a 0-certificate t in V^0 such that t covers the input at party i and t is incompatible at party i with at least $1/(2\alpha)$ of the current 1-certificates per bit needed to describe t at party i, for α as specified below. We shall see that the number of bits needed is $\mathrm{length}(t_i)+\lceil\log(2k_0^2k_1)\rceil+2$, so t must be incompatible with at least $(\mathrm{length}(t_i)+\lceil\log(2k_0^2k_1)\rceil+2)/(2\alpha)$ of the current 1-certificates at party i. Party i communicates such a certificate, if found, to the coordinator, in which case each exposed party is told this t_i and updates the set of current 1-certificates accordingly (the 1-certificates incompatible with t at party i are discarded). The next exposed party is now considered, in a round-robin fashion.

(2) If no 1-certificates can be discarded as just described, then each 0-certificate that contains the input will be incompatible with at least half of the current 1-certificates at nonexposed parties. The coordinator and each exposed party can recognize this situation, find all the parties accessed by a fraction of at least $1/(2k_0)$ of the current 1-certificates, and add these parties to the set of exposed parties. Now each 0-certificate that contains the input has one more accessed party exposed.

The communication bound is obtained as follows. Since each bit communicated with a given exposed party discards at least $1/(2\alpha)$ of the current 1-certificates, 2α bits must discard more than half of the current 1-certificates. By the bound in the lemma, this halving can be done at most $C_1+\lceil k_1\log k_0\rceil$ times before all 1-certificates have been discarded. Adding another 2α bits to ensure that the description of the last 0-certificate used to discard 1-certificates is not truncated, we obtain a $(C_1+\lceil k_1\log k_0\rceil+1)(2\alpha)$ bound on the communication with each exposed party. With α as defined below, we can check that α is indeed at least as large as the description of a certificate at a party, and that the communication bound in the statement of the theorem is satisfied.

We shall see below that at most $2k_0k_1$ parties are exposed at each phase, for a total of $2k_0^2k_1$ parties over the entire execution of the algorithm (since we need not expose parties at phase k_0+1). If this bound is maintained, then a 0-certificate t in V^0 at party i can be described with $\mathrm{length}(t_i)$ bits, plus an additional $\log(2k_0^2k_1)$ bits to identify i within

the set of current exposed parties. In communicating this information to each exposed party j, two additional bits are used: one bit is used by party j to tell the coordinator whether, after 1-certificates have been discarded according to t, there is some new 0-certificate t' that party j can use to discard 1-certificates; and one bit is sent back by the coordinator to tell party j whether it wants to use this new t' as the next 0-certificate to discard 1-certificates. Thus the communication of t_i to each party costs $\text{length}(t_i) + \lceil\log(2k_0^2k_1)\rceil + 2$ bits.

We choose $\alpha = (C_0 + k_0(\lceil\log(2k_0^2k_1)\rceil + 2))$. If no exposed party i can provide a 0-certificate t in V^0 that covers the input at party i and is incompatible with a fraction of at least $\rho_i = (\text{length}(t_i) + \lceil\log(2k_0^2k_1)\rceil + 2)/(2\alpha)$ of the current 1-certificates at party i, then every 0-certificate t in V^0 consistent with the input is incompatible with at most $\Sigma_i\rho_i \leq 1/2$ of the current 1-certificates at exposed parties, where the sum is over the parties accessed in t (at most k_0 of them). Hence every 0-certificate t in V^0 consistent with the input must be incompatible with at least half of the current 1-certificates at nonexposed parties, as claimed.

Since at most k_1 parties are accessed by a single 1-certificate, the sum over all parties of the fraction of current 1-certificates that access them is k_1, and so the number of parties accessed by a fraction of at least $1/(2k_0)$ of these current certificates is at most $2k_0k_1$. This proves the bound on the number of exposed parties added at each phase, completing the proof. □

4. Conclusion and open problems. In this paper we studied communication complexity in a multiparty model. The approach is based on the two-party model and the decision tree model. Some results from the two basic models can be applied to our model. The main open problems are the existence of lower bounds in this model and the study of randomization. An intriguing question is whether a quadratic upper bound with $O(k_0k_1)$ parties accessed and with polynomial communication can be achieved. The study of other measures, such as the number of phases [7], [13], and of more general communication networks [18], has a special importance for understanding communication in distributed systems.

Acknowledgments. The authors are very grateful to Rüdiger Reischuk for his careful reading of the paper and his helpful comments.

REFERENCES

[1] L. ADELMAN, *Two theorems on random polynomial time*, Proc. 19th IEEE Foundations of Computer Science, 1978, pp. 75–83.

[2] A. V. AHO, J. D. ULLMAN, AND M. YANNAKAKIS, *On notions of information transfer in* VLSI *circuits*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 133–139.

[3] M. BLUM AND R. IMPAGLIAZZO, *Generic oracles and oracle classes*, Proc. 19th ACM Symposium on the Theory of Computing, 1987, pp. 118–126.

[4] L BABAI, N. NISAN, AND M. SZEGEDY, *Multiparty protocols and logspace-hard pseudorandom sequences*, Proc. 21th ACM Symposium on the Theory of Computing, 1989, pp. 1–11.

[5] A. CHANDRA, M. FURST, AND R. LIPTON, *Multiparty protocols*, Proc. 15th ACM Symposium on the Theory of Computing, 1983, pp. 94–99.

[6] D. DOLEV AND T. FEDER, *Multiparty communication complexity*, Proc. 30th IEEE Foundations of Computer Science, 1989, pp. 428–433.

[7] P. DURIS, Z. GALIL, AND G. SCHNITGER, *Lower bounds on communication complexity*, Proc. 16th ACM Symposium on the Theory of Computing, 1984, pp. 81–91.

[8] M. FÜRER, *The power of randomness for communication complexity*, Proc. 19th ACM Symposium on the Theory of Computing, 1987, pp. 178–181.

[9] B. HALSTENBERG AND R. REISCHUK, *On different modes of communication*, Proc. 20th ACM Symposium on the Theory of Computing, 1988, pp. 162–172.

[10] J. HARTMANIS AND L. H. HEMACHANDRA, *One-way functions, robustness, and non-isomorphism of* NP-*complete sets*, Tech. Rep. DCS TR86-796, Cornell University, Ithaca, NY, 1987.

[11] K. MEHLHORN AND E. M. SCHMIDT, *Las Vegas is better than determinism in* VLSI *and distributed computing*, Proc. 14th ACM Symposium on the Theory of Computing, 1982, pp. 330–337.

[12] N. NISAN, CREW PRAM*s and Decision Trees*, Proc. 21st ACM Symposium on the Theory of Computing, 1989, pp. 327–335.

[13] C. H. PAPADIMITRIOU AND M. SIPSER, *Communication complexity*, Proc. 14th ACM Symposium on the Theory of Computing, 1982, pp. 196–200.

[14] R. RIVEST AND S. VUILLEMIN, *On recognizing graph properties from adjacency matrices*, Theoret. Comput. Sci., 3 (1978), pp. 371–384.

[15] M. SNIR, *Lower bounds for probabilistic linear decision trees*, Theoret. Comput. Sci., 38 (1985), pp. 69–82.

[16] M. SAKS AND A. WIGDERSON, *Probabilistic Boolean decision trees and the complexity of evaluating game trees*, Proc. 27th IEEE Foundations of Computer Science, 1986, pp. 29–38.

[17] G. TARDOS, *Query complexity, or why is it difficult to separate* $\mathcal{NP}^A \cap \mathrm{co}\mathcal{NP}^A$ *from* $\mathcal{P}^A$ *by a random oracle* A?, 1988, manuscript.

[18] P. TIWARI, *Lower bounds on communication complexity in distributed computer networks*, J. ACM, 34 (1987), pp. 921–938.

[19] A. YAO, *Some complexity questions related to distributive computing*, Proc. 11th ACM Symposium on the Theory of Computing, 1979, pp. 209–213.

SIAM J. COMPUT.
Vol. 21, No. 5, pp. 896–908, October 1992

© 1992 Society for Industrial and Applied Mathematics
006

ON THRESHOLD CIRCUITS AND POLYNOMIAL COMPUTATION*

JOHN H. REIF† AND STEPHEN R. TATE†

Abstract. A *Threshold Circuit* consists of an acyclic digraph of unbounded fanin, where each node computes a threshold function or its negation. This paper investigates the computational power of Threshold Circuits. A surprising relationship is uncovered between Threshold Circuits and another class of unbounded fanin circuits which are denoted Finite Field $Z_{P(n)}$ Circuits, where each node computes either multiple sums or products of integers modulo a prime $P(n)$. In particular, it is proved that all functions computed by Threshold Circuits of size $S(n) \geq n$ and depth $D(n)$ can also be computed by $Z_{P(n)}$ Circuits of size $O(S(n) \log S(n) + nP(n) \log P(n))$ and depth $O(D(n))$. Furthermore, it is shown that all functions computed by $Z_{P(n)}$ Circuits of size $S(n)$ and depth $D(n)$ can be computed by Threshold Circuits of size $O((1/\epsilon^2)(S(n) \log P(n))^{1+\epsilon})$ and depth $O((1/\epsilon^5)D(n))$. These are the main results of this paper.

There are many useful and quite surprising consequences of this result. For example, an integer reciprocal can be computed in size $n^{O(1)}$ and depth $O(1)$. More generally, any analytic function with a convergent rational polynomial power series (such as sine, cosine, exponentiation, square root, and logarithm) can be computed within accuracy 2^{-n^c}, for any constant c, by Threshold Circuits of polynomial size and constant depth. In addition, integer and polynomial division, FFT, polynomial interpolation, Chinese Remaindering, all the elementary symmetric functions, banded matrix inverse, and triangular Toeplitz matrix inverse can be exactly computed by Threshold Circuits of polynomial size and constant depth. All these results and simulations hold for polytime uniform circuits. This paper also gives a corresponding simulation of logspace uniform $Z_{P(n)}$ Circuits by logspace uniform Threshold Circuits requiring an additional multiplying factor of $O(\log \log \log P(n))$ depth.

Finally, purely algebraic methods for lower bounds for $Z_{P(n)}$ Circuits are developed. Using degree arguments, a Depth Hierarchy Theorem for $Z_{P(n)}$ Circuits is proved: for any $S(n) \geq n$, $D(n) = O(S(n)^{c'})$ for some constant $c' < 1$, and prime $P(n)$ where $6(S(n)/D(n))^{D(n)} < P(n) \leq 2^n$, there exist explicitly constructible functions computable by $Z_{P(n)}$ Circuits of size $S(n)$ and depth $D(n)$, but provably not computable by $Z_{P(n)}$ Circuits of size $S(n)^c$ and depth $o(D(n))$ for any constant $c \geq 1$.

Key words. circuit complexity, threshold circuits, finite field circuits

AMS(MOS) subject classifications. 68Q25, 68Q40

1. Introduction. A *threshold$_k$ function* is a boolean function whose output is 1 depending on whether at least k of its inputs have value 1. For example, a *threshold$_5$ function* is defined to be 1 if at least 5 inputs are 1. A *Threshold Circuit* is a boolean circuit in which each node computes a threshold function or its negation, and the nodes have unbounded fanin.

Many basic physical devices such as transistors and neurons can be modeled as threshold devices. Since an individual neuron may have very high fanin, a Threshold Circuit is a natural model for a neural net. For reasons described below, we will be particularly concerned with bounded depth Threshold Circuits.

Certainly any massively parallel computing device that uses a large number of relatively slow components must have small computational depth on a given computation if the overall computation is to be fast. For example, the reaction time of the lower brain for many nontrivial behavioral and recognition responses is less than .5 seconds, whereas the synapse response time of most neurons of the brain is at least .005 seconds; therefore, the depth of these particular computations can be no more than 100. Nevertheless, in this small depth, many nontrivial functions are computed by the brain. Minsky and

*Received by the editors January 30, 1989; accepted for publication (in revised form) September 5, 1991. This research was supported by National Science Foundation grant CCR-8696134, by a grant from the Office of Naval Research contract ONR-N00014-87-K-0310, and by Air Force Office of Scientific Research contract AFSOR-87-0386.

†Department of Computer Science, Duke University, Durham, North Carolina 27706.

Papert were among the first investigators to observe the relationship between the lower brain and constant depth Threshold Circuits [15]. In particular, they developed a model for a learning device, known as a Perceptron, which is essentially a threshold circuit with constant depth.

There has been a considerable amount of renewed interest in models for the brain and for learning, and many of the recently proposed models are again essentially constant depth Threshold Circuits. Examples of these models include the Connectionist Models [5] and the Boltzmann Machine [1], [10]. Recently, Parberry and Schnitger proved that Boltzmann Machines can be simulated by constant depth Threshold Circuits [16].

This paper is a further theoretical investigation of bounded depth Threshold Circuits. In particular, we consider the following fundamental computational question: *What class of functions can be computed by bounded depth Threshold Circuits?*

This paper is organized as follows: In §2, we give definitions of Threshold and Finite Field Circuits. In §3, we give a precise statement of our results. In §4, we give a simulation of Threshold Circuits by Finite Field Circuits. In §5, we give simulations of polytime uniform Finite Field Circuits by polytime uniform Threshold Circuits, thus characterizing the functions computed by Threshold Circuits of depth $D(n)$ as a certain class of multivariate polynomial functions computed by Finite Field Circuits of depth $\Theta(D(n))$. In §6, we give similar simulation results for logspace constructible circuits. In §7 we prove a Hierarchy Theorem for size bounded Finite Field Circuits with increasing depth. In §8, we conclude the paper with some open problems, conjectures, and some comments on how our theoretical results on Threshold Circuits might be applied to the construction of parallel arithmetic VLSI chips and to biological studies of learning in neuron nets by interpolation.

2. Circuit definitions.

2.1. Circuits that compute boolean functions.

Fix a value domain Σ. A *function basis* is a set F of functions over domain Σ^k, for each $k \geq 0$. We assume a binary decoding function $\text{decode}_{n,n'} : \{0,1\}^n \to \Sigma^{n'}$ for decoding length n binary strings into n' values in Σ, and an encoding function $\text{encode}_{m',m} : \Sigma^{m'} \to \{0,1\}^m$, for binary encoding vectors of m' values in Σ into binary strings of length m. We will define circuits that take n binary values as input, decode these inputs to an n'-tuple of values in Σ, make a computation using the functions in F, and then encode the outputs in binary.

A *circuit* C_n over function basis F is an oriented, acyclic digraph with a list of *input nodes* $v_1, \cdots, v_{n'}$, a list of *output nodes* $u_1, \cdots, u_{m'}$, and a k-adic function in F labeling each noninput node with fanin $k \geq 0$. Given a binary input string $(x_1, \cdots, x_n) \in \{0,1\}^n$, we decode the input as $\text{decode}_{n,n'}(x_1, \cdots, x_n) = (y_1, \cdots, y_{n'})$ where $(y_1, \cdots, y_{n'}) \in \Sigma^{n'}$, and assign each input node v_i a value $\text{val}(v_i) = y_i \in \Sigma$, for $i = 1, \cdots, n'$. For each other node w, with say k predecessors $w_1, \cdots, w_k$, we recursively assign w a value $\text{val}(w) = f(\text{val}(w_1), \cdots, \text{val}(w_k)) \in \Sigma$, where $f \in F$ is the k-adic function that labels node w. C_n finally outputs the binary string given by $\text{encode}_{m',m}(\text{val}(u_1), \cdots, \text{val}(u_{m'})) \in \{0,1\}^m$ (where the output length m is fixed for the circuit C_n). Thus C_n computes a boolean function from $\{0,1\}^n$ to $\{0,1\}^m$.

We shall allow the circuits considered in this paper to have arbitrary fanin. The *size* of circuit C_n is the number of edges of the circuit. The *depth* of circuit C_n is the length of the longest path from any input node to an output node. A *circuit family* is an infinite list of circuits $\mathbf{C} = (C_1, C_2, \cdots, C_n, \cdots)$, where C_n has n binary inputs. $\mathbf{C}$ computes a family of boolean functions $(f_1, f_2, \cdots, f_n, \cdots)$, where f_n is the function of n binary inputs computed by circuit C_n. Let $\mathbf{C}$ have *size complexity* $S(n)$ and simultaneous *depth*

complexity $D(n)$ if, $\forall n \geq 0$, circuit C_n has size $\leq S(n)$ and depth $\leq D(n)$.

Circuit family $\mathbf{C}$ is *polytime* (*logspace*) *uniform* if there exists a Turing machine M with $n^{O(1)}$ time bound ($O(\log n)$ space bound, respectively), such that given any $n \geq 1$ in unary, M constructs an encoding of circuit C_n.

2.2. Threshold circuits. A threshold function is a boolean function denoted $\delta_{k,\Delta} : \{0,1\}^k \to \{0,1\}$ such that

$$\delta_{k,\Delta}(x_1, \cdots, x_k) = \begin{cases} 1 & \text{if } \sum_{i=1}^{k} x_i \geq \Delta, \\ 0 & \text{otherwise} \end{cases}$$

for $x_1, \cdots, x_k \in \{0,1\}$. Let Th denote the set of all threshold functions and their negations. A *Threshold Circuit* is a circuit with function basis Th. Note that in the case of Threshold Circuits the value domain is $\Sigma = \{0,1\}$, so the number of input nodes is always the same as the number of boolean inputs, and *decode* and *encode* are simply the identity functions (i.e., no decoding of inputs or encoding of outputs is required). We let $\text{Th}(S(n), D(n))$ denote the collection of boolean function families computed by polytime uniform Threshold Circuits of size $O(S(n))$ and simultaneous depth $O(D(n))$. In addition, we will use the notation (logspace uniform) $\text{Th}(S(n), D(n))$ to denote the corresponding function families computed by logspace uniform Threshold Circuits. Note that with this notation, the class of all functions computed by Threshold Circuits having polynomial size and constant depth is $\text{Th}(n^{O(1)}, 1)$.

2.3. Finite field circuits. Let p be a prime number. For finite field circuits, the value domain Σ is Z_p, the finite field modulo p. We will let FZ_p denote the set of functions consisting of k-adic addition and multiplication taken modulo p for each $k \geq 1$, as well as a constant function giving value y, for each $y \in Z_p$. A (Finite Field) Z_p Circuit C_n is a circuit over function basis FZ_p. Let $b = \lfloor \log p \rfloor$. Given binary inputs $x_1, \cdots, x_n \in \{0,1\}$, we decode these inputs into $n' = \lceil n/b \rceil$ integer values $\text{decode}_{n,n'}(x_1, \cdots, x_n) = (y_1, \cdots, y_{n'})$, where the value $y_i \in Z_{2^b}$ is the number with binary encoding $x_{(i-1)b+1}, x_{(i-1)b+2}, \cdots, x_{\min(n,ib)}$. Note that the decoding of binary inputs yields only numbers in the range $\{0, 1, \cdots, 2^b - 1\} \subseteq Z_p$. The circuit C_n then makes a computation over FZ_p as described in §2.1. If $u_1, \cdots, u_{m'}$ are the output nodes, then we encode the output as $\text{encode}_{m',m}(\text{val}(u_1), \cdots, \text{val}(u_{m'})) = B_1 \cdots B_{m'}$, where B_i is the $t_i = \min(m - b(i-1), b)$ bit binary encoding of the integer residue of $\text{val}(u_i) \bmod 2^{t_i}$. We let $Z_{P(n)}(S(n), D(n))$ denote the collection of boolean function families computed by polytime uniform $Z_{P(n)}$ Circuit families $\mathbf{C} = (C_1, C_2, \cdots, C_n, \cdots)$, where each C_n is a $Z_{P(n)}$ Circuit with size $O(S(n))$ and simultaneous depth $O(D(n))$. We will use the additional notation (logspace uniform) $Z_{P(n)}(S(n), D(n))$ to denote the corresponding function families computed by logspace uniform $Z_{P(n)}$ Circuits.

3. Statement of results. In the following we let $P(n)$, $S(n)$, and $D(n)$ be any positive functions of n such that $S(n) \geq n$, and $P(n)$ is prime for all n.

We will first give a simulation of (polytime uniform) Threshold Circuits by (polytime uniform) Finite Field Circuits.

THEOREM 3.1. *If $S(n) \leq P(n) \leq n^{O(1)}$ for all n, then*

$$\text{Th}(S(n),\, D(n)) \subseteq Z_{P(n)}(S(n) \log S(n) + nP(n) \log P(n), D(n)).$$

Note. Theorem 3.1 also holds for logspace uniform circuits.

Next we will give a simulation of (polytime uniform) Finite Field Circuits by (polytime uniform) Threshold Circuits.

THEOREM 3.2. $Z_{P(n)}(S(n), D(n)) \subseteq Th((1/\epsilon^2)(S(n)\log P(n))^{1+\epsilon}, (1/\epsilon^5)D(n))$.

The proof of Theorem 3.2 requires that we build up families of Threshold Circuits for the basic problems of multiplication, iterated sum, and iterated product. The costliest problem we encounter is iterated product, and this is solved using techniques introduced for integer division [2], [8], [18].

As a consequence of Theorem 3.2, we have the following.

COROLLARY 3.3. *Suppose an analytic function $f(x)$ has a convergent Taylor Series Expansion of form*

$$f(x) = \sum_{n=0}^{\infty} c_n (x - x_0)^n$$

over an interval $|x - x_0| \le \epsilon$, where $0 < \epsilon < 1$, and the coefficients are rationals $c_n = \frac{a_n}{b_n}$, where a_n, b_n are integers of magnitude $\le 2^{n^{O(1)}}$. Then polytime uniform Threshold Circuits of polynomial size and simultaneous constant depth (i.e., a function in $\mathrm{Th}(n^{O(1)}, 1)$) can compute $f(x)$ over this interval within accuracy 2^{-n^c} for any constant $c \ge 1$.

Note that Corollary 3.3 follows directly from Theorem 3.2 since a Finite Field $Z_{P(n)}$ Circuit of size $n^{O(1)}$ and depth $O(1)$ with $P(n) = 2^{n^{O(1)}}$ can simulate the rational arithmetic required to approximately evaluate $f(x)$.

Corollary 3.3 implies (see [18]) that $\mathrm{Th}(n^{O(1)}, 1)$ contains a surprisingly rich class of elementary functions (which can be computed within accuracy 2^{-n^c}), including integer reciprocal, sine, cosine, exponential, logarithm, and square root, as well as exact computation of the following:

1. integer and polynomial quotient and remainder,
2. interpolation of rational polynomials,
3. banded matrix inverse, and
4. triangular Toeplitz matrix inverse.

These problems can all be efficiently reduced to integer products; also see [3], [4], [12], [18]. Theorems 3.1 and 3.2 yield the following characterization.

COROLLARY 3.4. *For $S(n) \le P(n) \le n^{O(1)}$,*

$$\bigcup_{c\ge 1} Z_{P(n)}(S(n)^c, D(n)) = \bigcup_{c \ge 1} Th(S(n)^c, D(n)).$$

For example, for $S(n) = n^{O(1)}$, $D(n) = O(1)$, $P(n) \le n^{O(1)}$, we get

$$Z_{P(n)}(n^{O(1)}, 1) = \mathrm{Th}(n^{O(1)}, 1).$$

In other words, the class of functions computed by polytime uniform $Z_{P(n)}$ Circuits of polynomial size and constant depth is exactly the same as the class of functions computed by polytime uniform Threshold Circuits of polynomial size and constant depth.

Next, we will give a simulation of logspace uniform Finite Field Circuits by logspace uniform Threshold Circuits.

THEOREM 3.5.

$$\begin{aligned}&(\text{logspace uniform})Z_{P(n)}(S(n), D(n))\\ &\qquad \subseteq (\text{logspace uniform})\mathrm{Th}((S(n)\log(P(n)))^{O(1)}, D(n)\log\log\log P(n)).\end{aligned}$$

The proof of Theorem 3.5 uses techniques developed by Reif for integer division by uniform boolean circuits of bounded fanin, polynomial size, and $O(\log n \log\log n)$ depth [18]. Theorem 3.5 implies that (logspace uniform) $\mathrm{Th}(n^{O(1)}, \log\log n)$ contains the various elementary functions listed above.

Finally, we derive some lower bound results for Finite Field Circuits using algebraic degree arguments.

THEOREM 3.6. *If $D(n) = O(S(n)^{c'})$ for some constant $c' < 1$, $D'(n) = o(D(n))$, and $6(S(n)/D(n))^{D(n)} < P(n) \le 2^n$, then there exists a function in $Z_{P(n)}(S(n), D(n))$ that is not in $\bigcup_{c\ge 1} Z_{P(n)}(S(n)^c, D'(n))$.*

Previously, Kung showed that degree bounded polynomials formed a hierarchy [13], but this did not immediately imply our result for $Z_{P(n)}$ Circuits.

4. Simulation of threshold circuits by finite field circuits. The key to our simulations will be the following.

LEMMA 4.1. *For prime p and any function $f : Z_p \to Z_p$ there is a (polytime and logspace uniform) Z_p Circuit of size $O(p \log p)$ and depth $O(1)$ which computes f.*

Proof. Any function $f : Z_p \to Z_p$ can be interpolated within Z_p at all p of its inputs, yielding a degree $p-1$ polynomial $p(x) = \sum_{i=0}^{p-1} c_i x^i$. In $O(p)$ size and $O(1)$ depth, we can compute x^{2^j} for $j = 0, \cdots, \lfloor \log p \rfloor$. From these values we can compute x^i for each $i = 1, \cdots, p-1$ in $O(p \log p)$ size and $O(1)$ depth. It follows that $f(x)$ is computable by a Z_p Circuit of size $O(p \log p)$ and depth $O(1)$. □

4.1. Proof of Theorem 3.1. Let C_n be a polytime uniform Threshold Circuit of n binary inputs $(x_1, \cdots, x_n) \in \{0,1\}^n$, where C_n has size $S(n)$ and depth $D(n)$. For any prime $p = P(n) \ge S(n)$, we will construct a Z_p Circuit C'_n that will also take n binary inputs $(x_1, \cdots, x_n) \in \{0,1\}^n$. Let $b = \lfloor \log p \rfloor$. By definition (see §2.3), C'_n must have $n' = \lceil \frac{n}{b} \rceil$ input nodes $v_1, \cdots, v_{n'}$, which are assigned integers $\mathrm{val}(v_1) = y_1, \cdots, \mathrm{val}(v_{n'}) = y_{n'}$, where $\mathrm{decode}_{n,n'}(x_1, \cdots, x_n) = (y_1, \cdots, y_{n'})$. The first difficulty we must overcome is to compute within C'_n the boolean encoding $x_{(i-1)b+1}, x_{(i-1)b+2}, \cdots, x_{\min(n,ib)} \in \{0,1\}$ of each integer y_i (i.e., these boolean values must be computed by C'_n from the y_i values using only addition and multiplication modulo p). By Lemma 4.1, there exists a polynomial $f_j(y)$ of degree $\le p-1$ which, when evaluated in Z_p, gives the boolean value of the jth bit of $y \in Z_p$, so each $x_{(i-1)b+j} = f_j(y_i)$ can be computed in C'_n using size $O(p \log p)$ and depth $O(1)$. The total size required here is $O(np \log p)$.

Next we must simulate in C'_n a threshold function $\delta_{k,\Delta}$ of k binary inputs, which we will denote $a_1, \cdots, a_k$. This can be done by first computing the sum $s = \sum_{i=1}^k a_i$ and then by finding the interpolating polynomial of degree $k-1$ that computes the function

$$\lambda_\Delta(s) = \begin{cases} 1 & s \ge \Delta, \\ 0 & s < \Delta. \end{cases}$$

This interpolating polynomial can be evaluated in size $O(k \log k)$ and depth $O(1)$. The negation of $\delta_{k,\Delta}$ can be computed in Z_p by a similar application of Lemma 4.1. This simulation of the threshold computations of C_n requires the Z_p Circuit C'_n to have size $O(S(n) \log S(n))$ and depth $O(D(n))$. Finally, if C_n has (boolean valued) output nodes $u_1, \cdots, u_m$, then we let C'_n have output nodes $u'_1, \cdots, u'_{m'}$ where $m' = \lceil \frac{m}{b} \rceil$. For $i = 1, \cdots, m'$ we compute the values $\mathrm{val}(u'_i) = \sum_{j=1}^{t_i} 2^j \mathrm{val}(u_{(i-1)b+j})$, where $t_i = \min(b, m - b(i-1))$, so $\mathrm{encode}_{m',m}(\mathrm{val}(u'_1), \cdots, \mathrm{val}(u'_{m'})) = (\mathrm{val}(u_1), \cdots, \mathrm{val}(u_m))$, and the (boolean) function computed by C'_n is exactly the same as the function computed by

C_n. The constructed $Z_{P(n)}$ Circuit C'_n has $O(S(n)\log S(n) + nP(n)\log P(n))$ size and $O(D(n))$ depth, and C'_n is polynomial time constructible, thus completing the proof of Theorem 3.1. □

Note that if C_n is logspace uniform, then C'_n is also logspace uniform.

5. Simulation of finite field circuits by threshold circuits.

5.1. Computing arithmetic using polytime constructible threshold circuits. The problem of finding the sum of a set of numbers is called the *iterated sum problem*. Pippenger has given a constant depth threshold circuit for multiplication, and the method used is the straightforward reduction to iterated sum (i.e., the "grade-school method" of multiplication) [17]. Looking at just the iterated sum circuit, we see that Pippenger's circuit for adding m values, each of n bits, has size $O(nm^2)$ and depth $O(1)$. In the following lemma, we show how to produce a constant depth circuit for iterated sum with smaller size.

LEMMA 5.1. *Given any constant ϵ satisfying $0 < \epsilon \leq 1$, there exists a circuit for computing the iterated sum of m numbers, each of n bits, (with $m \leq n^{O(1)}$) that has size $O(nm^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon})$.*

Proof. Since $m \leq n^{O(1)}$, it is trivial to show that the result of the iterated sum will have less than cn bits for some constant c.

To calculate the iterated sum, we build a computation tree with maximum fanout $\lfloor m^\epsilon \rfloor$ and m leaves. Placing the m input values at the leaves, computation proceeds toward the root of the tree with each internal node computing the sum of its children. After all computations, the root contains the sum of all m input values. It is easy to see that the desired tree has $O(m^{1-\epsilon})$ internal nodes, and a height of $O(\frac{1}{\epsilon})$. We use Pippenger's circuit at each internal node for a node size of $O(nm^{2\epsilon})$, so the total circuit size is $O(nm^{1+\epsilon})$. Since the depth of each node in the tree is constant, the total depth of the circuit is the same as the height of the tree or $O(\frac{1}{\epsilon})$. □

Using this result, we can also construct small size circuits for discrete Fourier transform. Let DFT_M denote the discrete Fourier transform of an M-vector.

LEMMA 5.2. *Given any constant ϵ satisfying $0 < \epsilon \leq 1$, we can construct a circuit for* $\mathrm{DFT}_M(a_0, a_1, \cdots, a_{M-1}) \bmod 2^N + 1$ *(where M and N are both powers of 2 and $M \leq N$) that has size $O(\frac{1}{\epsilon}MN^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^2})$.*

Proof. Since N and M are powers of 2, let $N = 2^n$ and $M = 2^m$. We will first show DFT_M exists in the ring Z_{2^N+1}. If we let $\omega = 2^{2N/M}$, then by taking $\omega^{M/2} = 2^N \equiv -1$ (mod $2^N + 1$) it is easy to see that ω is a principle Mth root of unity in Z_{2^N+1}. Also, since M is a power of 2, we know that M and $2^N + 1$ are relatively prime; therefore, M^{-1} exists in the ring. By these facts, the ring Z_{2^N+1} supports DFTs on M-vectors.

We introduce a new constant $\delta = (\sqrt{1+4\epsilon} - 1)/2$. We will construct a computation tree as we did in Lemma 5.1, but the fanout in this case will be $f = 2^{\lfloor m\delta \rfloor}$. Let $v_0, v_1, \cdots, v_{f-1}$ be the children of the root, and assume each child recursively computes the M/f-vector $\mathrm{val}(v_i) = (x_{i,0}, x_{i,1}, \cdots, x_{i,M/f-1}) = \mathrm{DFT}_{M/f}(a_i, a_{f+i}, \cdots, a_{M-f+i})$. Note that these vectors exist since ω^f is a principle M/fth root of unity, and $(M/f)^{-1}$ exists in Z_{2^N+1}. From these vectors we can produce the vector $(y_0, y_1, \cdots, y_{M-1}) = \mathrm{DFT}_M(a_0, a_1, \cdots, a_{M-1})$ by calculating

$$y_i = \sum_{j=0}^{f-1} \omega^j x_{j,i} \bmod 2^N + 1. \tag{1}$$

The proof of correctness for (1) is straightforward and is not included in this paper. Equation (1) is a simple modular iterated sum, since multiplication by powers of ω is

just a bit shift of zero cost. This process is repeated down the tree until there are fewer than f values in each node. In general, if we label the root as level 0, we are calculating DFT_{M/f^i} at each node of level i from its f children. By using the iterated sum circuit of Lemma 5.1 (the reduction mod 2^N+1 can be done after a nonmodular iterated sum with a single subtraction), we can do this in size $O((M/f^i)Nf^{1+\delta})$ for each node on level i. Since there are f^i nodes on level i, the total size for all nodes of that level is $O(MNf^{1+\delta})$. There are $O(\frac{1}{\delta})$ levels, so the total size of the circuit is $O(\frac{1}{\delta}MNf^{1+\delta})$. Since f is $O(N^\delta)$, the size can be written as $O(\frac{1}{\delta}MN^{1+\delta+\delta^2}) = O(\frac{1}{\epsilon}MN^{1+\epsilon})$. The depth of each level is $O(\frac{1}{\delta})$, so the total depth is $O(\frac{1}{\delta^2}) = O(\frac{1}{\epsilon^2})$. □

Using this circuit for discrete Fourier transform we can construct a constant depth multiplication circuit.

LEMMA 5.3. *Given any constant ϵ satisfying $0 < \epsilon \leq 0.6$, we can construct a circuit for multiplication of two N bit numbers that has size $O(\frac{1}{\epsilon}N^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^3})$.*

Proof. The circuits that we construct are actually for multiplying two N-bit numbers modulo 2^N+1, where N is a power of 2. For exact (nonmodular) multiplication of N' bit numbers, we use the same circuit with $N = 2^{\lceil \log N' \rceil + 1}$. It is easy to show that this will produce the exact answer.

We will denote the two input numbers by a and b, and their product by c. Since N is a power of 2, let $N = 2^n$, where n is an integer. Letting $m = 2^{\lfloor \epsilon n \rfloor}$, we can write any N-bit number a as an m-vector of blocks of $s = \frac{N}{m}$ bits, $a = (a_0, a_1, \cdots, a_{m-1})$; a_0 is the block of least significant bits. We can view this vector as a vector of polynomial coefficients and define the polynomial $A(x) = \sum_{i=0}^{m-1} a_i x^i$. Note that $A(2^s) = a$. Defining a polynomial for b in a similar way, the product polynomial $C(x) = A(x)B(x)$ will be such that $C(2^s) = c$.

We use discrete Fourier transforms for the polynomial multiplication, and since the product polynomial will have degree $2m-2$, we must calculate the transform of $2m$-vectors. (We could actually use wrapped convolutions on m-vectors, but nothing is gained over our asymptotic bounds.) Looking at the straightforward method of polynomial multiplication, it is easy to bound $\max_{0 \leq i < 2m}\{c_i\} < m2^{2s} < m(2^{2s}+1)$. Since m and $2^{2s}+1$ must be relatively prime, we can calculate the coefficients of $C(x)$ modulo both m and $2^{2s}+1$ and combine these results for the final answer modulo $m(2^{2s}+1)$. This ring includes as a subset the range of all possible results, so the result of these modular calculations is also the exact (nonmodular) answer. The calculations modulo m can be done using Lemma 5.1 and "grade-school multiplication," with a total size of $O(N^{1+\epsilon})$ as long as $\epsilon \leq 0.6$. We will now concentrate on the cost of the calculations modulo $2^{2s}+1$.

We will again use a divide-and-conquer tree with the root labeled as level 0. The fanout of the tree is $2m$, and it should be obvious that on level i we are computing products of $s_i = N\left(\frac{2}{m}\right)^i$ bit numbers. The $\mathrm{DFT}_{2m} \bmod (2^{2s_{i+1}}+1)$ required at this level can be done in size $O(\frac{1}{\epsilon}2m(2s_{i+1})^{1+\epsilon})$ by Lemma 5.2. On level i, there are $(2m)^i$ such DFTs to calculate, for a total size of $O(\frac{1}{\epsilon}4^{i+1}\left(\frac{2}{m}\right)^{(i+1)\epsilon}(2N)^{1+\epsilon})$. For sufficiently large N (and, therefore, m) we have $(\frac{m}{2})^\epsilon > 8$, so the size of level i can be simplified to $O(\frac{1}{\epsilon}\left(\frac{1}{2}\right)^i N^{1+\epsilon})$. Summing over all levels we have a total size of $O(\frac{1}{\epsilon}N^{1+\epsilon})$.

The depth of each level in the tree is $O(\frac{1}{\epsilon^2})$ by Lemma 5.2, so the total depth of our multiplication circuit is $O(\frac{1}{\epsilon^3})$. □

Note. The requirement that $\epsilon \leq 0.6$ can be relaxed to $\epsilon \leq 1$ by simply creating a new constant $\delta = \frac{\epsilon}{2}$ and absorbing the constant factor increase in depth into the big-Oh notation; however, this is clearly just a notational manipulation and not an algorithmic improvement.

The problem of Chinese Remaindering can be stated as follows: given m small primes $p_1, p_2, \cdots, p_m$ (actually, they only have to be pairwise relatively prime) and an n bit number a, calculate the residue of $a \bmod p_i$ for all $1 \le i \le m$. Conversely, given the residues modulo each of the primes $r_1, r_2, \cdots, r_m$, we would like to calculate the least positive a such that $a \equiv r_i \pmod{p_i}$ for all $1 \le i \le m$. We will only be interested in the case where $m \ge n$, and this fact simplifies the analysis.

LEMMA 5.4. *Given any constant ϵ satisfying $0 < \epsilon \le 0.6$, we can construct a circuit for Chinese Remaindering (in both directions) with size $O(\frac{1}{\epsilon^2} m^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^4})$.*

Proof. The method of Chinese Remaindering is taken straight from [8], using the multiplication circuit of Lemma 5.3. The proof of the size and depth of the circuit is also analogous to that found in [8], and is not included in this paper. □

The last basic problem we will look at is that of an iterated product over a finite field. An iterated product of m values $a_1, a_2, \cdots, a_m$ over the field Z_p is defined to be $\prod_{i=1}^{m} a_i \bmod p$.

LEMMA 5.5. *Given any constant ϵ satisfying $0 < \epsilon \le 1$, we can construct a circuit for iterated product of m numbers over the field Z_p with size $O(\frac{1}{\epsilon^2}(m \log p)^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^5})$.*

Proof. Define a new constant $\delta = \frac{\epsilon}{5}$. We will perform the iterated product in a tree similar to the tree used for iterated sum. The tree will have fanout m^δ and will perform an iterated product of m^δ values in Z_p at each node. The iterated product at each node is computed by performing a Chinese Remainder step, followed by calculating the iterated product over each of the smaller fields (using discrete logs, iterated sum, and powering), and finally a Chinese Remaindering step to recover the full result. This produces the exact iterated product, and by multiplying by an $m^\delta \log p$ bit approximation to $1/p$, we can find the residue modulo p.

To insure there is no loss of information, we must be sure that $\prod p_i$ is greater than the maximum possible result. Specifically, we must insure that $\prod_{i=1}^{s} p_i > p^{m^\delta}$. By basic number theoretic results, we can achieve this with $s \le p_s = \Theta(m^\delta \log p)$. Obviously, $s > \log p$, so the condition of Lemma 5.4 is satisfied, and we may construct the required Chinese Remaindering circuit with size $O(\frac{1}{\delta^2} m^{2\delta} (\log p)^{1+\delta})$ and depth $O(\frac{1}{\delta^4})$.

After performing the initial Chinese Remaindering step, we must perform an iterated product over each of the p_i. Since for all prime p_i, Z_{p_i} is a cyclic group, there is a (not necessarily unique) generator — call it g_i — that generates the entire group. Let $f_i(x) = g_i^x$; due to the fact that g_i is a generator, this function is one-to-one and onto over $Z_{p_i}^*$. We make tables for $f_i(x)$ and $f_i^{-1}(x)$, each of size $O(p_i \log p_i)$. Within a particular field, there must be tables for all m^δ input values, so the total size taken up by tables for p_i is $O(m^\delta p_i \log p_i)$.

The iterated product is calculated by taking the discrete logarithm of all input values ($f_i^{-1}(x)$, above), performing the iterated sum of these values modulo $p_i - 1$, then raising the generator to the resulting power in Z_{p_i} (this is just $f_i(x)$, above). This is a fairly common method of performing iterated product (see, for example, [2]). The only part we have not examined here is the iterated sum. By Lemma 5.1, we can calculate the exact iterated sum of m^δ numbers, each of $\log p_i$ bits, in size $O(m^{\delta+\delta^2} \log p_i)$ and depth $O(\frac{1}{\delta})$. With an $m^\delta \log p_i$ bit approximation to $(1/(p_i - 1))$, we can reduce this exact result to the result modulo $p_i - 1$ with a single multiplication. By Lemma 5.3, this takes size $O(\frac{1}{\delta} m^{\delta+\delta^2} (\log p_i)^{1+\delta})$ and depth $O(\frac{1}{\delta^3})$; therefore, the total complexity of calculating the iterated product of m^δ numbers modulo p_i is $O(\frac{1}{\delta^2} m^{2\delta} p_i (\log p_i)^{1+\delta})$ size and $O(\frac{1}{\delta^4})$ depth.

Since this must be done for all s prime fields, the total size complexity of iterated product of m^δ numbers is s times the above value, plus the cost of Chinese Remaindering. Using the upper bounds for s and p_i, the total size is $O(\frac{1}{\delta}m^{5\delta}(\log p)^{1+2\delta})$ and the total depth is $O(\frac{1}{\delta^4})$. With an $m^\delta \log p$ bit approximation to $(1/p)$, we can reduce this result (the exact iterated product) modulo p. The complexity of this multiplication is negligible compared to the rest of the circuit.

All the above results are for one node of the tree. Summing over all nodes and rewriting in terms of ϵ, the total size is $O(\frac{1}{\delta^2}m^{1+5\delta}(\log p)^{1+2\delta}) = O(\frac{1}{\epsilon^2}(m \log p)^{1+\epsilon})$, and the depth is $O(\frac{1}{\epsilon^5})$. □

Note. All Threshold Circuit families considered in this section can easily be seen to be constructed in polynomial time.

5.2. Proof of Theorem 3.2. Now we are ready to prove Theorem 3.2. Consider any polytime uniform $Z_{P(n)}$ Circuit C_n with size $S(n)$ and depth $D(n)$. We wish to simulate C_n by a Threshold Circuit $\hat{C}_n$. We will precompute an $S(n) \log P(n)$ bit approximation of the reciprocal of $P(n)$ so that a residue computation modulo $P(n)$ node of fanin k can be done by just $O(k)$ additions and multiplications on $O(\log P(n))$ bit binary numbers, followed by a residue computation using an $O(k \log P(n))$ bit approximation to $\frac{1}{P(n)}$; therefore, each iterated sum or iterated product required at a node of C_n can be done by applying Lemmas 5.1 and 5.2 using only size $O(\frac{1}{\epsilon^2}(k \log P(n))^{1+\epsilon})$ and depth $O(\frac{1}{\epsilon^5})$. The total size of the Threshold Circuit $\hat{C}_n$ is $O(\frac{1}{\epsilon^2}(S(n) \log P(n))^{1+\epsilon})$, and the depth is $O(\frac{1}{\epsilon^5}D(n))$; furthermore, the circuit family $\hat{C}$ is constructible in polynomial time, completing the proof of Theorem 3.2. □

6. Log space uniform threshold circuit simulation of arithmetic and finite field circuits. Let $a_1, \cdots, a_m \in Z_{2^n}$. Let $D(m,n)$ be the depth required to compute $\prod_{i=1}^m a_i \bmod (2^n+1)$ using a (logspace uniform) Threshold Circuit of size $(mn)^{O(1)}$.

LEMMA 6.1. $D(m,n) \le D(m, O(mn)^{1/2}) + O(1)$.

Proof. We use a reduction of Reif from iterated product to discrete Fourier transform [18]. Assume, without loss of generality, that n is a power of 2, and let $\hat{n} = O(mn)^{1/2}$ also be a power of 2. Given $a_1, \cdots, a_m \in Z_{2^n}$, we let $a_{i,j}$ (for $i = 1, \cdots, m$ and $j = 0, \cdots, \hat{n}-1$) be integers in $Z_{2^{n/\hat{n}}}$ such that $a_i = \sum_{j=0}^{\hat{n}-1} a_{i,j} 2^{jn/\hat{n}}$. We first compute in the vector $(g_{i,0}, \cdots, g_{i,\hat{n}-1}) = DFT((a_{i,0}, 2a_{i,1}, \cdots, 2^{\hat{n}-1}a_{i,\hat{n}-1})^T)$ for $i = 1, \cdots, m$. By Lemma 5.2, we can easily compute these DFTs in polynomial size and constant depth. For $k = 0, \cdots, \hat{n}-1$ compute in (logspace uniform) Th$((m\hat{n})^{O(1)}, D(m,\hat{n}))$ the iterated product $e_k = \prod_{i=1}^m g_{i,k} \bmod (2^{\hat{n}}+1)$. Finally, compute in (logspace uniform) Th$((nm)^{O(1)}, 1)$ the vector $(f_0, 2f_1, \ldots, 2^{\hat{n}-1}f_{\hat{n}-1}) = DFT^{-1}((e_0, \cdots, e_{\hat{n}-1})^T)$ and output $\sum_{i=0}^{\hat{n}-1} f_i 2^{in/\hat{n}} = \prod_{i=1}^m a_i \bmod (2^n+1)$. The total depth is the depth of the recursion plus a constant amount, as stated in the lemma. □

LEMMA 6.2. $D(m,n) \le O(\log m \log\log n / \log n)$.

Proof. For any ϵ, $0 < \epsilon < \frac{1}{2}$, we can compute the iterated product of m integers by first computing the $\lceil m/n^\epsilon \rceil$ iterated products of n^ϵ integers, and repeating this $\lceil \frac{\log m}{\epsilon \log n} \rceil$ times, we get $D(m,n) \le \lceil \frac{\log m}{\epsilon \log n} \rceil (D(n^\epsilon, n) + O(1))$. Applying Lemma 6.1 and this recurrence a constant number of times, we get

$$D(n^\epsilon, n) \le D(n^\epsilon, n^{1/2}) + O(1) \le D(n^{\epsilon/2}, n^{1/2}) + O(1).$$

Finally, applying the above recurrence $\log\log n$ times, we get $D(n^\epsilon, n) \leq O(\log\log n)$. Hence

$$D(m,n) = O\left(\frac{\log m}{\log n}\right) O(\log\log n),$$

which is the bound claimed in the lemma. □

6.1. Proof of Theorem 3.5. Note that Lemma 6.2 implies that iterated product of $n^{O(1)}$ integers with n bits each is in (logspace uniform) $\mathrm{Th}(n^{O(1)}, \log\log n)$. Since computing the n bit approximation of the reciprocal of an n bit number reduces to simply computing the iterated sum of n iterated products of size n, we can also compute residues modulo a number with n bits in (logspace uniform) $\mathrm{Th}(n^{O(1)}, \log\log n)$. Theorem 3.5 immediately follows, since we must compute residues, iterated sums, and iterated products of $n = O(\log P)$ bit numbers. □

7. Lower bounds. The degree of a multivariable polynomial $f(y_1, \cdots, y_k)$ is the maximum sum of the powers of the variables appearing in any term (monomial) of $f(y_1, \cdots, y_k)$.

LEMMA 7.1. *Suppose $f(y_1, \cdots, y_k)$ is a nonzero polynomial of degree d over a finite field Z_p, and A is a subset of Z_p of size σ. If $\sigma > d$, then $\exists(a_1, \cdots, a_k) \in A^k$ such that $f(a_1, \cdots, a_k) \neq 0$.*

Proof. The proof is by induction on k. For the basis case $k = 1$, we have $f(y_1)$, which is only a single variable polynomial. It is well known that any nonzero polynomial $f(x)$ of degree d over any field can have at most d zeros in the field (see, for example, [6]), and since $\sigma > d$, at least one $a \in A$ must give a nonzero value for $f(a)$.

We make the induction hypothesis that the lemma holds for all polynomials with $< k$ variables. Since $f(y_1, \cdots, y_k)$ is nonzero, $\exists(u_1, \cdots, u_k) \in (Z_p)^k$ such that $f(u_1, \cdots, u_k) \neq 0$. Hence $f(u_1, y_2, \cdots, y_k) = f'(y_2, \cdots, y_k)$ is not a zero polynomial, and by the induction hypothesis, $\exists(a_2, \cdots, a_k) \in A^{k-1}$ such that $f'(a_2, \cdots, a_k) = f(u_1, a_2, \cdots, a_k) \neq 0$. Let $g(y_1) = f(y_1, a_2, \cdots, a_k)$. $g(x)$ is clearly a nonzero polynomial, so by the basis step there is an $a_1 \in A$ such that $g(a_1) \neq 0$, and we have constructed $(a_1, a_2, \cdots, a_k) \in A^k$ such that $f(a_1, \cdots, a_k) \neq 0$. □

Note. A similar lemma for polynomial identity testing in infinite fields was proved by Ibarra and Moran [11].

7.1. Proof of Theorem 3.6. Fix any positive integer functions $S(n)$ and $D(n)$, where $D(n) = O(S(n)^{c'})$ for some constant $c' < 1$, and $S(n) \geq n$. Now consider a sequence of primes $\{P(1), P(2), \cdots, P(n), \cdots\}$, where $6(S(n)/D(n))^{D(n)} < P(n) \leq 2^n$. We will construct a family of $Z_{P(n)}$ circuits $\mathbf{C} = (C_1, C_2, \cdots, C_n, \cdots)$ of size $S(n)$ and depth $D(n)$. In particular, we let $v_1, \cdots, v_{n'}$ be the input nodes of C_n, where $n' = \lceil n/b \rceil$ and $b = \lfloor \log P(n) \rfloor \leq n$. We also let $w_0 = v_1$ denote the first input node. Each level $L = 1, \cdots, D(n)$ of C_n consists of a single "product" node w_L with $\lfloor S(n)/D(n) \rfloor$ edges entering w_L from node w_{L-1}, so that $\mathrm{val}(w_L)$ is the $\lfloor S(n)/D(n) \rfloor$ power of $\mathrm{val}(w_{L-1})$; $w_{D(n)}$ is the unique output node of C_n. Let $y_1 = \mathrm{val}(v_1), \cdots, y_{n'} = \mathrm{val}(v_{n'})$ be the input values, and let $\vec{y} = (y_1, \cdots, y_{n'})$. We have constructed C_n of size $\leq S(n)$ and depth $D(n)$ so that its output is the $d_n = (\lfloor S(n)/D(n) \rfloor)^{D(n)}$ degree polynomial $f_n(\vec{y}) = \mathrm{val}(w_{D(n)}) = (y_1)^{d_n}$. Note, however, that by definition, C_n gets decoded input integers $y_1, \cdots, y_{n'}$ only over input domain Z_{2^b}, whereas the computation is over the entire Finite Field $Z_{P(n)}$. Furthermore, the binary encoded output value is the residue $f_n(\vec{y}) \bmod 2^b$.

Next consider any $Z_{P(n)}$ circuit family $\mathbf{C}' = (C'_1, C'_2, \cdots, C'_n, \cdots)$, where C'_n has size $S(n)^c$, for some constant $c \geq 1$, and simultaneous depth $D'(n) = o(D(n))$. We can

assume without loss of generality that C'_n has only a single output node, which computes value $g_n(\vec{y})$, where $\vec{y} = (y_1, \cdots, y_{n'})$ are the decoded integer values of its input nodes. Again note that $g_n(\vec{y})$ has only input domain Z_{2^b}. We wish to show that there exists some $\vec{y} \in (Z_{2^b})^{n'}$ such that $f_n(\vec{y}) \neq g_n(\vec{y}) \bmod 2^b$. Observe that $g_n(\vec{y})$ is of degree $\leq \prod_{L=1}^{D'(n)} e_L$, where e_L is the number of edges of C'_n entering nodes of level L. This product form is maximized when each $e_L = S(n)^c/D'(n)$, and since $\sum_{L=1}^{D'(n)} e_L = S(n)^c$, we get an upper bound on the maximum possible degree of $g_n(\vec{y})$ as $(S(n)^c/D'(n))^{D'(n)}$.

Since $D'(n) = o(D(n))$, we have for infinitely many n, and any constant c,

$$\begin{aligned} d_n &= (\lfloor S(n)/D(n) \rfloor)^{D(n)} \\ &\geq (S(n)/(2D(n)))^{D(n)} > S(n)^{(1-c')D(n)} \\ &> S(n)^{cD'(n)} > (S(n)^c/D'(n))^{D'(n)} \geq \deg(g_n(\vec{y})). \end{aligned}$$

Fix some such n. For this value of n, by the above derivation $d_n > \deg(g_n(\vec{y}))$ and $d_n \leq (S(n)/D(n))^{D(n)} < P(n)/6$, so by Lemma 7.1 there exists some $\vec{y} \in (Z_{P(n)})^{n'}$ such that $f_n(\vec{y}) \neq g_n(\vec{y})$. However, this does not prove Theorem 3.6 because we must actually show there exists some $\vec{y} \in (Z_{2^b})^{n'}$ such that $f_n(\vec{y}) \neq g_n(\vec{y}) \bmod 2^b$.

We define a new function $h_n(\vec{y})$ by the equation

$$h_n(\vec{y}) = \left(f_n(\vec{y}) - g_n(\vec{y}) - 2^b\right)\left(f_n(\vec{y}) - g_n(\vec{y})\right)\left(f_n(\vec{y}) - g_n(\vec{y}) + 2^b\right).$$

Note that if $f_n(\vec{y}) = g_n(\vec{y}) \bmod 2^b$ for all $\vec{y} \in (Z_{2^b})^{n'}$, then $h_n(\vec{y}) = 0$ for all inputs $\vec{y} \in (Z_{2^b})^{n'}$.

The degree of $h_n(\vec{y})$ is easily seen to be $3d_n$, and it is also obvious that $h_n(\vec{y})$ is not identically zero. Let $A = Z_{2^b}$, and since we know that

$$\text{degree}(h_n(\vec{y})) = 3d_n < 3\left(\frac{S(n)}{D(n)}\right)^{D(n)} < \frac{P(n)}{2} < |A|,$$

we can use Lemma 7.1 to see that $h_n(\vec{y}) \neq 0$ for at least one n'-tuple $(a_1, a_2, \cdots, a_{n'}) \in (Z_{2^b})^{n'}$. Theorem 3.6 follows immediately. □

8. Conclusions.

8.1. Threshold circuits for arithmetic units. Division is by far the most costly operation for Arithmetic Units. Our polynomial size, constant depth Threshold Circuits for arithmetic indicate that Threshold Circuits might be quite useful in highly parallel Arithmetic Units for integer division and trigonometric computations. It is an interesting question as to whether a high fanin threshold gate can be manufactured in a reasonably small area on silicon chips. Constant fanin threshold gates are in fact used in current NMOS and CMOS technologies. In theory, fanin k threshold gates can be constructed so that with sufficient area (growing no more than quadratically with k) these gates can be driven in unit time. In particular, Mead and Conway describe how to construct tally circuits (for k input threshold) in VLSI with total area $O(k^2)$ and time $O(1)$ for moderate k using pass transistors [14, pp. 78–80]. The Microelectronics Center of North Carolina is investigating the use of new microelectronic devices that may be used for Threshold gates with large fanin. If this is feasible in practice, then VLSI Arithmetic Units might be designed using Threshold Circuits to run much faster than currently possible (i.e., compared with the standard bounded fanin boolean logic gates of conventional VLSI).

8.2. On learning and interpolation in neural networks. Our positive results concerning Threshold Circuits (in particular, Theorem 3.2 and Corollary 3.3) show that Threshold Circuits of polynomial size and constant depth can compute high accuracy approximations to a large class of multivariate rational polynomials, and, furthermore, can interpolate rational polynomials with a constant number of variables. Learning by algebraic interpolation appears to be appropriate in certain constrained cases such as low level vision [7], and would likely be much more efficient than previously proposed methods for learning (such as found in [1] and [9]), which are essentially brute force. Nevertheless, even making the apparently reasonable assumption that certain portions of the lower brain act essentially as Threshold Circuits of constant depth does not necessarily imply that the lower brain is wired so as to compute approximations or interpolations of multivariate polynomials. However, our theoretical results do provide strong evidence of the *feasibility* of neuron nets that evaluate and interpolate such polynomial functions.

A neural biologist might, for example, make experimental tests to verify this by using a computer to monitor input-output response functions of neuron nets. Specifically, the lower brain very rapidly provides feedback control for certain muscles; this control appears to be smooth and nonlinear. Such easily observable responses would appear to be ideal to monitor and to interpolate. By using known randomized multivariate polynomial identity tests, such as those of Ibarra and Moran [11], we can, with very high likelihood, verify that the input-output response of a neuron net is a specific interpolated multivariate polynomial.

8.3. Lower bound conjectures. Finally, we make two lower bound conjectures concerning Threshold Circuits.

CONJECTURE 8.1. *For* $D'(n) = o(D(n))$, *there exists an* $f \in \bigcup_{c\geq 1} \mathrm{Th}(n^c, D(n))$ *that is not in* $\bigcup_{c\geq 1} \mathrm{Th}(n^c, D'(n))$.

Let DETERMINANT be the problem "given an $n \times n$ matrix A with 0, 1 elements, compute the determinant of A."

CONJECTURE 8.2. DETERMINANT *is not contained in* $\mathrm{Th}(n^c, 1)$ *for any constant* c.

REFERENCES

[1] D. H. ACKLEY, G. E. HINTON, AND T. J. SEJNOWSKI, *A learning algorithm for Boltzmann machines*, Cognitive Sci., 9 (1985), pp. 147–169.

[2] P. W. BEAME, S. A. COOK, AND H. J. HOOVER, *Log depth circuits for division and related problems*, SIAM J. Comput., 15 (1986), pp. 994–1003.

[3] D. BINI, *Parallel solution of certain Toeplitz linear systems*, SIAM J. Comput., 13 (1984), pp. 268–276.

[4] D. BINI AND V. PAN, *Fast parallel algorithms for polynomial division over arbitrary field of constants*, Note Interna, Dipartimento di Informatica, Università di Pisa, Pisa, Italy, 1984.

[5] J. A. FELDMAN AND D. H. BALLARD, *Connectionist models and their properties*, Cognitive Sci., 6 (1982), pp. 205–254.

[6] J. B. FRALEIGH, *A First Course in Abstract Algebra*, Addison-Wesley, Reading, MA, 1982.

[7] W. E. L. GRIMSON, *From Images to Surfaces: A Computational Study of the Human Early Visual System*, MIT Press, Cambridge, MA, 1981.

[8] J. HASTAD AND T. LEIGHTON, *Division in* $O(\log n)$ *depth using* $O(n^{1+\epsilon})$ *processors*, 1986, unpublished note.

[9] G. E. HINTON, T. SEJNOWSKI, AND D. A. ACKLEY, *Boltzmann machines: constraint satisfaction networks that learn*, Tech. Report CMU-CS-84-119, Carnegie Mellon University, Pittsburgh, PA, 1984.

[10] J. J. HOPFIELD, *Neural networks and physical systems with emergent collective computational abilities*, Proc. Nat. Acad. Sci. USA, 79 (1982), pp. 2554–2558.

[11] O. H. IBARRA AND S. MORAN, *Probabilistic algorithms for deciding equivalence of straight-line programs*, J. ACM, 30 (1983), pp. 217–228.

[12] H. JUNG, *On probabilistic tape complexity and fast circuits for matrix inversion problems*, in Proceedings of the 11th Colloquium on Automata, Languages, and Programming, Springer-Verlag LNCS, Vol. 172, 1984, pp. 281–291.

[13] H. T. KUNG, *New algorithms and lower bounds for the parallel evaluation of certain rational expressions and recurrences*, J. ACM, 23 (1976), pp. 252–261.

[14] C. A. MEAD AND L. A. CONWAY, *Introduction to* VLSI *Systems*, Addison-Wesley, Reading, MA, 1980.

[15] M. L. MINSKY AND S. PAPERT, *Perceptrons: An Introduction to Computational Geometry*, MIT Press, Cambridge, MA, 1968.

[16] I. PARBERRY AND G. SCHNITGER, *Parallel computation with threshold functions*, in Proceedings of Conference on Structure in Complexity Theory, Springer-Verlag LNCS, Vol. 223, 1986, pp. 272–290.

[17] N. PIPPENGER, *The complexity of computations by networks*, IBM J. Res. Dev., 31 (1987), pp. 235–243.

[18] J. H. REIF, *Logarithmic depth circuits for algebraic functions*, SIAM J. Comput., 15 (1986), pp. 231–242.

SIAM J. COMPUT.
Vol. 21, No. 5, pp. 909–926, October 1992

© 1992 Society for Industrial and Applied Mathematics
007

GENERALIZING THE CONTINUED FRACTION ALGORITHM TO ARBITRARY DIMENSIONS*

BETTINA JUST†

Abstract. This paper presents for the first time a higher-dimensional continued fraction algorithm (abbreviated "cfa") that produces diophantine approximations of more than linear goodness. On input $x_1, \cdots, x_{n-1} \in \mathbf{R}$, it produces vectors $(p_1^{(k)}, \cdots, p_{n-1}^{(k)}, q^{(k)}) \in \mathbf{Z}^n$, $k = 1, 2, \cdots$, such that

$$\max_{1 \leq i \leq n-1} \left| x_i - \frac{p_i^{(k)}}{q^{(k)}} \right| \leq \frac{\|x\| \cdot \operatorname{const}(n)}{|q^{(k)}|^{1+1/(2n(n-1))}}.$$

By a theorem of Dirichlet, there is no algorithm that replaces the term $1/(2n(n-1))$ by a term bigger than $1/(n-1)$. The higher-dimensional cfa's analyzed so far do not achieve better than $\max_{1 \leq i \leq n-1} |x_i - p_i^{(k)}/q^{(k)}| \leq o(1)/|q^{(k)}|$. The $o(1)$ term decreases with k but is not known to be related with $q^{(k)}$.

Other properties of the cfa are also generalized by the algorithm. On input $x_1, \cdots, x_{n-1}$ it starts with the standard basis of $\mathbf{Z}^n$ and then constructs by performing elementary basis transformations a sequence $(\mathcal{B}^{(k)})_k$ of bases of $\mathbf{Z}^n$. The sequence $(\mathcal{B}^{(k)})_k$ is finite if and only if the numbers $x_1, \cdots, x_{n-1}, 1$ are $\mathbf{Z}$-linearly dependent; a linear dependence is found in case of existence. The maximal distance between the vectors of $\mathcal{B}^{(k)}$ and the straight line $(x_1, \cdots, x_{n-1}, 1)\,\mathbf{R}$ tends to zero exponentially fast in k. For each k, the above-mentioned vector $(p_1^{(k)}, \cdots, p_{n-1}^{(k)}, q^{(k)})$ is the first vector of basis $\mathcal{B}^{(k)}$.

The algorithm is a variant of an algorithm for the integer relation problem presented in [G. Bergman, *Notes on Ferguson and Forcade's Generalized Euclidean Algorithm*, preprint, Univ. California, Berkeley, 1980] and analyzed in [J. Hastad, B. Just, J. Lagarias, and C. P. Schnorr, *SIAM J. Comput.*, 18 (1989), pp. 859–881]. The bound on the goodness of the diophantine approximations is proven with a "parallel induction" technique.

Key words. continued fraction algorithm, diophantine approximation, integer relation

AMS(MOS) subject classifications. 11A55, 11J70, 68Q25

1. Introduction. The *continued fraction algorithm* (cfa) is one of the fundamental mathematical algorithms. Its underlying computational model is the unit cost model, that is, one step is an arithmetic operation $+, -, *, /$, a trunc $\lfloor . \rfloor$ to the next lower integer, or a comparison $\geq$ among real numbers. The cfa accepts as input one arbitrary real number x_1 and outputs a sequence of bases of $\mathbf{Z}^2$. Some fundamental properties of these bases are in the list that follows. For a detailed presentation and for the proofs of the properties mentioned, we refer to the literature (e.g., [22]).

In 1868 Jacobi [20] considered generalizations of the cfa. An *n-dimensional cfa* accepts as input real numbers $x_1, \cdots, x_{n-1}$ and outputs a sequence of bases of $\mathbf{Z}^n$. It obtains each basis from the previous one by performing a sequence of elementary basis transformations. (A *basis* of $\mathbf{Z}^n$, $n \geq 1$, is an ordered set $\{b_1, \cdots, b_n\} \subset \mathbf{Z}^n$ such that $\sum b_i \mathbf{Z} = \mathbf{Z}^n$. An *elementary basis transformation* transforms one basis of $\mathbf{Z}^n$ to another either by interchanging two basis vectors or by adding an integer multiple of one basis vector to another basis vector.)

One desires that the following four properties of the cfa carry over to higher dimensions.

Approximation of the straight line. The sequence $(\{b_1^{(k)}, \cdots, b_n^{(k)}\})_k$ of bases of $\mathbf{Z}^n$ produced on input $x_1, \cdots, x_{n-1}$ should fulfill

$$\max_{1 \leq i \leq n} \operatorname{dist}(b_i^{(k)}, x\,\mathbf{R}) \to_k 0$$

*Received by the editors June 5, 1989; accepted for publication (in revised form) July 23, 1991.

†Johann Wolfgang Goethe-Universität Frankfurt/Main, Fachbereich Mathematik, Robert-Mayer-Strasse 6-10, D-6000 Frankfurt, Federal Republic of Germany.

(if the sequence is infinite). Here and in what follows, the notation $x := (x_1, \cdots, x_{n-1}, 1)$ is used [5].

Integer relations. A vector $m = (m_1, \cdots, m_n) \in \mathbf{Z}^n \setminus \{0\}$ is called *integer relation for* $y = (y_1, \cdots, y_n) \in \mathbf{R}^n$ if $\sum_{i=1}^n m_i y_i = 0$. The cfa stops, if and only if the input x_1 is rational, that is, if and only if x fulfills an integer relation. A higher dimensional cfa should do so too (and should also find integer relations, if they exist).

Diophantine approximations. Vectors $b = (p_1, \cdots, p_{n-1}, q) \in \mathbf{Z}^n$ such that $\max_{1 \leq i \leq n-1} |x_i - p_i/q|$ is "small" are called (*simultaneous*) *diophantine approximations* for $x_1, \cdots, x_{n-1}$. For arbitrary given denominator q one can find trivially nominators $p_1, \cdots, p_{n-1}$ such that $\max_{1 \leq i \leq n-1} |x_i - p_i/q| \leq 1/(2|q|)$ holds. The following is a theorem of Dirichlet [10]: "For arbitrary $x_1, \cdots, x_{n-1} \in \mathbf{R}$ there exist infinitely many $(p_1, \cdots, p_{n-1}, q) \in \mathbf{Z}^n$ such that $\max_{1 \leq i \leq n-1} |x_i - p_i/q| \leq 1/\left(|q|^{1+1/(n-1)}\right)$."

The bound is sharp in the sense that the right side of the inequation cannot be replaced by $c/|q|^s$, where $s > 1 + 1/(n-1)$ and c and s are constants [6].

If $b = (p, q) \in \mathbf{Z}^2$ is a basis vector of the cfa on input x_1, then $|x_1 - p/q| \leq 1/|q|^2$ holds. For higher dimensions it is not known whether there exit entire bases of diophantine approximations fulfilling the Dirichlet bound (or some other nontrivial bound). So, one desires that some of the basis vectors are "good" (in some sense) diophantine approximations for the input numbers.

Periodicity. A sequence $(\mathcal{B}_k)_{k \in \mathbf{N}}$ of bases of $\mathbf{Z}^n$ is *periodic* if there exist $m_0, s \in \mathbf{N}$ and an $n \times n$ matrix T with integer entries, such that for all $r \in \mathbf{N}$ and $m \geq m_0$

$$[\mathcal{B}_m] \cdot T^r = [\mathcal{B}_{m+rs}].$$

Here $[\mathcal{B}]$ denotes the matrix with the basis vectors of $\mathcal{B}$ as columns. The sequence of bases produced by the cfa on input x_1 is periodic if and only if the field extension $\mathbf{Q}(x_1)$ of $\mathbf{Q}$ is of degree 2. One might desire that the sequence of bases produced by a higher dimensional cfa on input $x_1, \cdots, x_{n-1}$ is periodic if and only if the field extension $\mathbf{Q}(x_1, \cdots, x_{n-1})$ of $\mathbf{Q}$ is of degree n.

Since Jacobi's paper, higher dimensional cfas were proposed by Poincaré, Minkowski, Perron, Brun, Payley and Ursell, Rosser, Szekeres, and others [3], [5], [8], [16], [28]–[31], [36]. However, none of these algorithms has been proven to fulfill one of the four properties above. Only results for specific inputs are known.

In several papers since 1979, Bergman, Ferguson, and Forcade [4], [11]–[14] presented variations of an algorithm for the integer relation problem. It is the first known ideally convergent algorithm. It was analyzed by Hastad, Just, Lagarias, and Schnorr [17]. For any $\varepsilon > 0$, after $O(n^3(n + \log 1/\varepsilon))$ arithmetical steps on real numbers either it finds an integer relation for x or it proves that the Euclidean length of each integer relation for x is larger than $1/\varepsilon$. Babai, Just, and auf der Heide [2] showed that the parameter ε cannot be omitted: In a very powerful model of computation there is no algorithm that detects $\mathbf{Z}$ linear dependence in finite time.

It is natural to ask whether the Bergman–Ferguson–Forcade algorithm produces good diophantine approximations, and how about periodicity? Nothing is known about the latter. The former is implicit in the ideal convergence of the algorithm: Every ideally convergent algorithm produces diophantine approximations $p_1^{(k)}, \cdots, p_{n-1}^{(k)}, q^{(k)}$ such that $\max_{1 \leq i \leq n-1} |x_i - p_i^{(k)}/q^{(k)}| \leq o(1)/|q^{(k)}|$, where the $o(1)$ term decreases with k (cf. Claim 24 of this paper). However, it is not known how it is related to $q^{(k)}$.

The present paper for the first time presents a higher-dimensional continued frac-

tion algorithm that produces approximations with

$$\max_{1\le i\le n-1}\left|x_i-\frac{p_i^{(k)}}{q^{(k)}}\right|\le\frac{1}{|q^{(k)}|^{1+o(1)}}.$$

The bound is proven with a parallel induction technique. The algorithm is a variant of the Bergman–Ferguson–Forcade algorithm. On input $x_1,\cdots,x_{n-1}$ it starts with the standard basis of $\mathbf{Z}^n$ and then constructs, by performing elementary basis transformations, a sequence $(\mathcal{B}^{(k)})_k$ of bases of $\mathbf{Z}^n$. The sequence $(\mathcal{B}^{(k)})_k$ is finite if and only if the numbers $x_1,\cdots,x_{n-1},1$ are $\mathbf{Z}$ linearly dependent; a linear dependence is found in case of existence. The maximal distance between the vectors of $\mathcal{B}^{(k)}$ and the straight line $(x_1,\cdots,x_{n-1},1)\,\mathbf{R}$ tends to zero exponentially in k. For each k, the preceding vector $p_1^{(k)},\cdots,p_{n-1}^{(k)},q^{(k)}$ is the first vector of basis $\mathcal{B}^{(k)}$.

The paper is organized as follows. Section 2 contains mathematical facts and algorithmical techniques. In §3 the algorithm is presented and its performance is stated as a theorem, which is proven in §4. Section 5 contains remarks on the algorithm.

2. Mathematical and algorithmic preliminaries. We start with some lattice theory. Let $a_1,\cdots,a_s\in\mathbf{R}^d$ be linearly independent. Then the set $L(a_1,\cdots,a_s):=\sum a_i\,\mathbf{Z}$ of all integer linear combinations of the a_i is called *lattice spanned by* $a_1,\cdots,a_s$. The ordered set $\{a_1,\cdots,a_s\}$ is called the *basis* of the lattice, and the number s is the *dimension* of the lattice. The dimension is unique, the basis is not: For each unimodular $s\times s$ matrix T, the columns of the matrix $[a_1,\cdots,a_s]\cdot T$ also form a basis of the lattice. Moreover, all bases of $L(a_1,\cdots,a_s)$ are obtained in this way [9]. (A matrix T is *unimodular* if it has integer entries and $|\det T|=1$.) So the number

$$\det(L):=|\det[a_1,\cdots,a_s]^t\cdot[a_1,\cdots,a_s]|^{1/2}$$

is unique for the lattice and is called the *determinant* of L. Here, $[\cdots]^t$ denotes the transposed matrix.

The purpose of lattice basis reduction theory is to select "reduced" bases from the bases of the lattice. A reduced basis must consist of "short" vectors that are pairwise "fairly orthogonal." For the dimension 2 Gauss [15] defined reduced bases and presented an algorithm that constructs a reduced basis from an arbitrary given one. For higher dimensions, however, the definition of reduced bases is not canonical. Proposals were made by Hermite, Korkine and Zolotareff, Minkowski, and others [19], [23], [28].

In 1982 Lenstra, Lenstra, and Lovász presented the first polynomial time algorithm to construct a reduced lattice basis from an arbitrary given one. They invented a new definition of "reduced" that will be used in the present paper. Their algorithm starts with a lattice basis and then performs two kinds of basis transformations (size reduction steps and exchange steps) until a reduced basis is reached. The algorithm has been widely applied [1], [18], [21], [24]–[27], [33], [34].

Now let $\{a_1,\cdots,a_s\}$ be a lattice basis. For all i,j such that $1\le j\le i\le s$, we denote by $a_i(j)$ the orthogonal projection of a_i to the orthogonal complement $\ll a_1,\cdots,a_{j-1}\gg^{\perp}$ of the linear space spanned by $a_1,\cdots,a_{j-1}$. The $a_i(j)$ situation is illustrated in Fig. 2.1.

For all $i\in\{1,\cdots,s\}$ we also write a_i^* instead of $a_i(i)$; the a_i^* are an orthogonal system. Now for all $1\le i,j\le s$, we denote by $\mu_{i,j}$ the jth coefficient of a_i in this system; thus,

$$a_i=a_i^*+\sum_{j=1}^{i-1}\mu_{i,j}a_j^*. \tag{1}$$

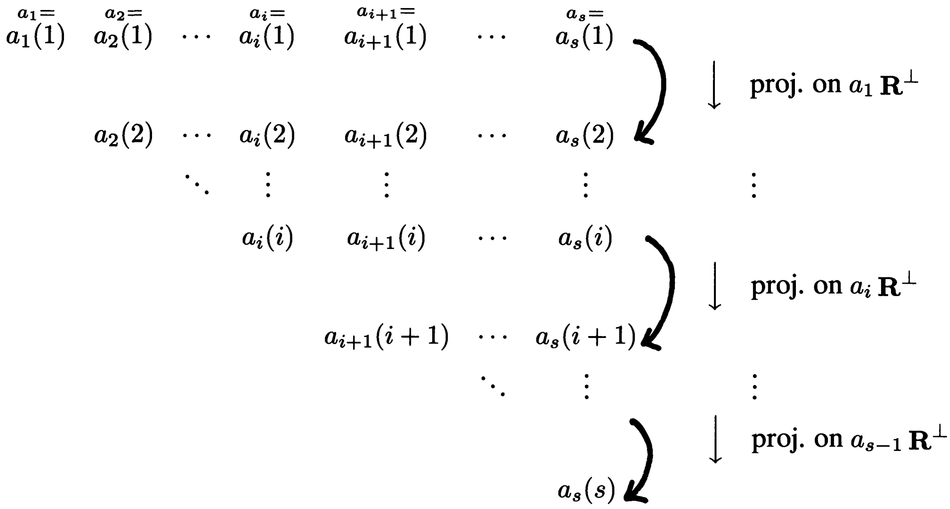


FIG. 2.1. *Projection of* a_i.

Moreover, we observe

$$\det(L(a_1, \cdots, a_s)) = \prod_{i=1}^{s} \|a_i^*\|, \tag{2}$$

where $\|.\|$ denotes the Euclidean norm.

Now, according to [27] we call a basis $\{a_1, \cdots, a_s\}$ *reduced* if it fulfills (3) and (4):

$$\|a_i^*\|^2 \leq 2 \cdot \|a_{i+1}^*\|^2 \quad \text{for all } i \in \{1, \cdots, s-1\}, \tag{3}$$

$$|\mu_{i,j}| \leq 1/2 \quad \text{for all } 1 \leq i < j \leq s. \tag{4}$$

LEMMA 1 [27]. *If the basis* $\{a_1, \cdots, a_s\}$ *is reduced, then*

$$\|a_1^*\|^2 \leq 2^{s-1} \cdot \|v\|^2 \quad \textit{for all } v \in L(a_1, \cdots, a_s) \setminus \{0\}, \tag{5}$$

$$\|a_1^*\| \leq 2^{(s-1)/4} \cdot \det(L(a_1, \cdots, a_s))^{1/s}. \tag{6}$$

We return to the task of generalizing the cfa to arbitrary dimensions. Let $x_1, \cdots, x_{n-1} \in \mathbf{R}$ be the input numbers, and denote the vector $(x_1, \cdots, x_{n-1}, 1)$ by x. For each vector $b \in \mathbf{R}^n$ we denote by πb the projection of b to the orthogonal complement $x\,\mathbf{R}^\perp$ of x. So the distance between b and the straightline $x\,\mathbf{R}$ is $\|\pi b\|$.

Now let $\{b_1, \cdots, b_n\}$ be a basis of $\mathbf{Z}^n$, for example, the standard basis that we take to begin with. We want to perform elementary basis transformations in order to get bases that more closely approximate $x\,\mathbf{R}$. To this end, we treat $\{\pi b_1, \cdots, \pi b_n\}$ as if it were a lattice basis to be reduced. Certainly it is not, since $\pi b_1, \cdots, \pi b_n$ are linearly dependent. Nevertheless, we perform size reduction steps and exchange steps. These steps provide elementary basis transformations on $b_1, \cdots, b_n$ that decrease $\pi b_1, \cdots, \pi b_n$ and, therefore, the distance of the basis vectors to the straight line.

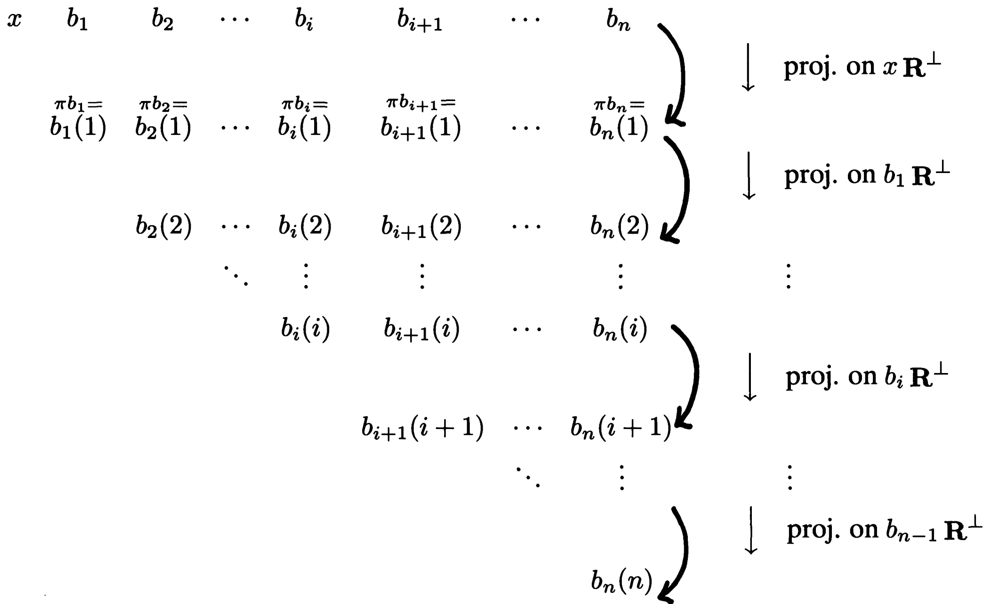


FIG. 2.2. *Projection of b_i.*

For $x, b_1, \cdots, b_n$ and all $1 \le j \le i \le n$, we denote by $b_i(j)$ the projection of b_i to the orthogonal complement $\ll x, b_1, \cdots, b_{j-1} \gg^\perp$ of the space spanned by $x, b_1, \cdots, b_{j-1}$. Figure 2.2 illustrates the $b_i(j)$ situation.

The vectors $x, b_1^*, \cdots, b_n^*$ form an orthogonal system; one b_i^* is the zero vector.

LEMMA 2 [17].

(a) *Denote by $\lambda(x)$ the Euclidean length of the shortest integer relation for x if one exists; otherwise, define $\lambda(x) := \infty$. Then for each basis $\{b_1, \cdots, b_n\}$ of $\mathbf{Z}^n$*

$$\lambda(x) \ge \frac{1}{\max_{1\le i\le n} \|b_i^*\|}. \tag{7}$$

(b) *Define* $[c_1, \cdots, c_n] := ([b_1, \cdots, b_n]^t)^{-1}$. *If $b_n^* \ne 0$, then c_n is an integer relation for x.*

Remark. Equation (7) implies that each ideally convergent algorithm solves the integer relation problem: The algorithm on input $x_1, \cdots, x_{n-1}, 1$ approximates the straight line $x\,\mathbf{R}$ with bases of $\mathbf{Z}^n$. This cannot be achieved if there is an integer relation for x, hence the algorithm will fail in that case. This "failure" proves the $\mathbf{Z}$-linear dependence of $x_1, \cdots, x_{n-1}, 1$.

By Lemma 2(b) we may assume from now on that $b_n^* = 0$, since otherwise we have found an integer relation and are finished. For $i \in \{1, \cdots, n\}$, we denote by $\mu_{i,j}$ the coordinates of b_i with respect to $x, b_1^*, \cdots, b_{n-1}^*$, thus,

$$b_i = \mu_{i,0} \cdot x + b_i^* + \sum_{j=1}^{i-1} \mu_{i,j} \cdot b_j^*, \tag{8}$$

(9) $$\|\pi b_i\|^2 = \|b_i^*\|^2 + \sum_{j=1}^{i-1} |\mu_{i,j}|^2 \cdot \|b_j^*\|^2.$$

Our aim is to decrease $|\mu_{i,j}|$ and $\|b_i^*\|$ for $1 \leq j < i \leq n$. To this end, size reduction steps and exchange steps are performed. Both transform one basis of $\mathbf{Z}^n$ to another.

A *size reduction step* replaces b_i by $b_i - \lceil \mu_{i,j} \rfloor \cdot b_j$ for one pair (i,j) with $1 \leq j < i \leq n$. Here $\lceil . \rfloor$ denotes the nearest integer. This achieves $|\mu_{i,j}^{new}| \leq 1/2$. All b_i^* and all μ's except the $\mu_{i,l}$ with $l \leq j$ are unchanged. By performing size reduction steps in a nested loop for all $i = 1$ to n and for all $j = i - 1$ down to 1, one achieves $|\mu_{i,j}| \leq 1/2$ for all $1 \leq j < i \leq n$.

An *exchange step* $i \leftrightarrow i+1$ (for $1 \leq i \leq n-1$) is allowed only if $\|b_i^*\|^2 \geq 2 \cdot \|b_{i+1}^*\|^2$. It first performs—if $|\mu_{i+1,i}| > 1/2$—a size reduction step to achieve $|\mu_{i+1,i}| \leq 1/2$. Then it interchanges b_i and b_{i+1}. This changes b_i^*, b_{i+1}^*, and $\mu_{r,l}$ with $\{r,l\} \cap \{i, i+1\} \neq \emptyset$.

An exchange step $i \leftrightarrow i+1$ achieves

(10) $$\|b_i^{*new}\|^2 \leq \tfrac{3}{4} \cdot \|b_i^{*old}\|^2,$$

since $\|b_i^{*new}\|^2 = \|b_{i+1}(i)^{old}\|^2 = \|b_{i+1}^{*old}\|^2 + |\mu_{i+1,i}| \cdot \|b_i^{*old}\|^2$. Since b_{i+1}^{*new} is a projection of b_i^{*old}, we have

(11) $$\|b_{i+1}^{*new}\| \leq \|b_i^{*old}\|.$$

The following claim is obvious if $b_{i+1}^{*old} = 0$; otherwise, it holds because the exchange step leaves the lattice $L(b_i(i)^{old}, b_{i+1}(i)^{old})$, and hence its determinant, fixed:

(12) $$\|b_i^{*old}\| \cdot \|b_{i+1}^{*old}\| = \|b_i^{*new}\| \cdot \|b_{i+1}^{*new}\|.$$

Equations (10), (11), and (12) imply the following.

Result 3. Each exchange step decreases $D := \prod_{i=1}^{n-1} \|b_i^*\|^{2(n-i)}$ by at least a factor of $\frac{3}{4}$.

Now we consider algorithms that perform size reduction steps and (allowed) exchange steps as long as $b_n^* = 0$. Such algorithms have the following readily checked properties.

CLAIM 4. $\max \|b_i^*\|$ *never increases.*

CLAIM 5. $\min_{i \neq n} \|b_i^*\|$ *decreases only (but not necessarily) at exchange steps* $n-1 \leftrightarrow n$.

CLAIM 6. *Exchange steps* $n-1 \leftrightarrow n$ *decrease* $\prod_{i=1}^{n-1} \|b_i^*\|$ *by at least a factor of* $\frac{1}{2}$; *the other exchange steps and the size reduction steps leave* $\prod_{i=1}^{n-1} \|b_i^*\|$ *unchanged.*

The Bergman–Ferguson–Forcade algorithm and the generalized cfa presented in the present paper perform size reduction steps and exchange steps in order to decrease the $\|b_i^*\|$ and to achieve $|\mu_{i,j}| \leq \frac{1}{2}$. By (9) the distance between the b_i and the straight line $x\,\mathbf{R}$ becomes small. The algorithms differ in where exchange steps are performed. The Bergman–Ferguson–Forcade algorithm always performs exchange steps $i \leftrightarrow i+1$ at the position i where $2^i \cdot \|b_i^*\|^2 = \max_{1 \leq j \leq n} 2^j \cdot \|b_j^*\|^2$. Our algorithm performs arbitrary exchange steps with the restriction that exchange steps $n-1 \leftrightarrow n$ are performed only if no others are allowed. However, the basic idea for both algorithms is the same.

Exchange steps bring short b_i^* "upwards" and long b_i^* "downwards" in Fig. 2.2. If $b_n^* \neq 0$, an integer relation is output as a result of Lemma 2(b). Otherwise, at least an exchange step $n-1 \leftrightarrow n$ is allowed, which shortens a "long" b_{n-1}^*. If b_{n-1}^* becomes short enough, an exchange step $n-2 \leftrightarrow n-1$ is allowed. This shortens b_{n-2}^*, and so on.

3. The algorithm. In this section we present a higher-dimensional cfa. On input $x_1, \cdots, x_{n-1} \in \mathbf{R}$ it produces a sequence $(\mathcal{B}^{(k)})_k$ of bases of $\mathbf{Z}^n$. The performance of the algorithm will be stated as a theorem, which will be proven in the next section.

ALGORITHM

Higher-dimensional cfa $(x_1, \cdots, x_{n-1})$.

Step 1 (initialization):
$x := (x_1, \cdots, x_{n-1}, 1)$;
for $i \in \{1, \cdots, n\}$ let b_i be the ith unit vector;
for all $1 \leq j \leq i \leq n$ compute b_i^* and $\mu_{i,j}$;

Step 2 (exchange steps of the first basis vectors):
while $\exists i < n-1$ such that $\|b_i^*\|^2 \geq 2 \cdot \|b_{i+1}^*\|^2$ do
(i) $b_{i+1} := b_{i+1} - \lceil \mu_{i+1,i} \rfloor \cdot b_i$;
(ii) interchange b_i and b_{i+1};
(iii) update b_i^*, b_{i+1}^* and the μ's;

Step 3 (size reduction):
for $i = 2$ to n do
for $j = i-1$ downto 1 do
(i) $b_i := b_i - \lceil \mu_{i,j} \rfloor \cdot b_j$;
(ii) update μ's;
output $\mathcal{B}^{(k)} := \{b_1, \cdots, b_n\}$;

Step 4 (exchange step $n-1 \leftrightarrow n$):
interchange b_{n-1} and b_n;
update μ's and b_{n-1}^*, b_n^*;
if $b_n^* \neq 0$
then $[c_1, \cdots, c_n] := ([b_1, \cdots, b_n]^{-1})^t$;
output integer relation c_n for x;
else goto 2. □

Remarks. The notation $\mathcal{B}^{(k)}$ stands for the basis $\mathcal{B}$efore the kth exchange step $n-1 \leftrightarrow n$. We note that when entering in step 4 we have $|\mu_{n,n-1}| \leq \frac{1}{2}$, thus an exchange step $n-1 \leftrightarrow n$ interchanges the last two basis vectors without size reduction.

THEOREM 7. *On input $x_1, \cdots, x_{n-1} \in \mathbf{R}$ the algorithm starts with the standard basis of $\mathbf{Z}^n$ and performs a sequence of elementary basis transformations. It outputs a subsequence $(\mathcal{B}^{(k)})_k$ of the obtained bases.*

(a) *If the numbers $x_1, \cdots, x_{n-1}, 1$ are $\mathbf{Z}$-linearly independent, the sequence $(\mathcal{B}^{(k)})_k$ is infinite. Otherwise, the algorithm stops after finitely many steps and outputs an integer relation c_n for x such that*

$$\|c_n\|^2 \leq 2^{n-2} \cdot \lambda(x_1, \cdots, x_{n-1}, 1)^2.$$

(b) *If $\delta := dist(x\,\mathbf{R}, b_1^{(k)})$, then the basis $\mathcal{B}^{(k)}$ is obtained after at most $O(n^4(n + \log 1/\delta))$ elementary basis transformations. Moreover, each basis $\mathcal{B}^{(k)}$ fulfills*

$$\max_{1 \leq i \leq n} \text{dist}(x\,\mathbf{R}, b_i^{(k)}) \leq \sqrt{n-1} \cdot 2^{(n-1)/2} \cdot 2^{-(k-1)/(n-1)}.$$

(c) *The first basis vector $b_1^{(k)} := (p_1, \cdots, p_{n-1}, q)$ of each basis $\mathcal{B}^{(k)}$ fulfills*

$$\max_{1 \leq i \leq n-1} |x_i - \frac{p_i}{q}| \leq \frac{\|x\| \cdot 2^{(n+2)/4}}{|q|^{1+1/(2n(n-1))}}.$$

4. Analysis of the algorithm. The purpose of this section is to prove Theorem 7. Parts (a) and (b) are proven with methods similar to those used in [17]. Part (c) is proven with a parallel induction technique that (as far as the author knows) was not used before in lattice theory or in the context of cfa's. Before starting the proof, we note the following:

Note 8. The basis $\{\pi b_1^{(k)}, \cdots, \pi b_{n-1}^{(k)}\}$ is a reduced lattice basis for all k. Thus,

$$2^{n-1} \cdot \|b_{n-1}^{(k)}\| = \max_{1 \le i \le n-1} 2^i \cdot \|b_i^{(k)}\| \quad \text{for all } k. \tag{13}$$

We shall first prove part b of Theorem 7, then part (a), and then part (c).

4.1. Proof of Theorem 7(b). We know from Result 3 that each exchange step decreases the number $D = \prod_{i=1}^{n-1} \|b_i^*\|^{2(n-i)}$ by at least a factor of $\frac{3}{4}$. At the beginning we have $D \le 1$. The basis $\mathcal{B}^{(k)}$ fulfills $\|b_i^{(k)*}\|^2 \le 2\|b_{i+1}^{(k)*}\|^2$ for all $1 \le i \le n-2$; otherwise, step 2 would not have been finished. So we have

$$D \ge \prod_{i=1}^{n-1} \left(2^{1-i} \cdot \|b_1^{(k)*}\|^2\right)^{n-i}$$

for the basis $\mathcal{B}^{(k)}$. This shows that at most $O(n^2(n + \log 1/\|b_1^{(k)*}\|))$ exchange steps are performed until $\mathcal{B}^{(k)}$ is output. Since each exchange step is followed by at most $O(n^2)$ size reduction steps, the claim on the number of elementary transformations is proven.

To bound $\operatorname{dist}(x\,\mathbf{R}, b_i^{(k)})$ we define, for each intermediary basis $\{b_1, \cdots, b_n\}$ of the algorithm (not only for the output bases $\mathcal{B}^{(k)}$) and for each $\varepsilon > 0$, the measure D_ε by

$$D_\varepsilon(b_1, \cdots, b_n) := \prod_{i=1}^{n-1} \max\{\varepsilon, \|b_i^*\|\}.$$

We first show the following.

CLAIM 9. *D_ε never increases throughout the algorithm.*

Proof. Since size reduction steps leave the b_i^* and thus D_ε fixed, it suffices to show that exchange steps do not increase D_ε.

We consider the exchange step $i \leftrightarrow i+1$, which transforms the basis $\{b_1^{old}, \cdots, b_n^{old}\}$ to the basis $\{b_1^{new}, \cdots, b_n^{new}\}$. With the notation $\alpha(b) := \max\{\varepsilon, \|b\|\}$ we have

$$\frac{D_\varepsilon(b_1^{new}, \cdots, b_n^{new})}{D_\varepsilon(b_1^{old}, \cdots, b_n^{old})} = \frac{\alpha(b_i^{new*}) \cdot \alpha(b_{i+1}^{new*})}{\alpha(b_i^{old*}) \cdot \alpha(b_{i+1}^{old*})}.$$

Therefore, we have to show

$$\frac{\alpha(b_i^{new*}) \cdot \alpha(b_{i+1}^{new*})}{\alpha(b_i^{old*}) \cdot \alpha(b_{i+1}^{old*})} \le 1. \tag{14}$$

To prove (14), we observe that α is increasing in $\|b_i^*\|$. Now, if $\alpha(b_i^{new*}) = \varepsilon$, (14) holds, since $\|b_i^{*old}\| \ge \|b_{i+1}^{*new}\|$. If $\alpha(b_i^{new*}) = \|b_i^{new*}\|$ and $\alpha(b_{i+1}^{new*}) = \|b_{i+1}^{new*}\|$, (14) follows from (12). If $\alpha(b_i^{new*}) = \|b_i^{new*}\|$ and $\alpha(b_{i+1}^{new*}) = \varepsilon$, (14) follows from (10). This proves (14) and, thus, Claim 9.

Now, let $0 < \varepsilon \le 1$ be fixed. At the beginning $D_\varepsilon \le 1$, and $D_\varepsilon \ge \varepsilon^{n-1}$ always. Moreover, as long as $\max_{1 \le i \le n-1} \|b_i^{(k)*}\|^2 \ge 2^{n-1} \cdot \varepsilon^2$, we have $\|b_{n-1}^{(k)*}\| \ge 2\varepsilon$; hence,

the interchange step $n-1 \leftrightarrow n$ applied on $\mathcal{B}^{(k)}$ in step 4 decreases D_ε by at least a factor of 2. So by Claim 9 and Claim 4 we have $\max_{1\le i\le n-1} \|b_i^{(k)*}\|^2 < 2^{n-1}\cdot\varepsilon^2$ for all $k \ge 1+(n-1)\cdot\log 1/\varepsilon$. Hence, $\max_{1\le i\le n} \mathrm{dist}(b_i^{(k)}, x\,\mathbf{R}) \le \sqrt{n-1}\cdot 2^{(n-1)/2}\cdot\varepsilon$ holds for these k. This implies part (b) of our theorem.

4.2. Proof of Theorem 7(a). We first show that each execution of step 2 takes only a finite amount of time. It suffices to show that the while loop is performed only finitely often.

We denote by $\mathcal{A}^{(k)} = \{a_1^{(k)}, \cdots, a_n^{(k)}\}$ the basis of the algorithm After the kth exchange step $n-1 \leftrightarrow n$. The basis $\mathcal{A}^{(k)}$ is obtained from $\mathcal{B}^{(k)}$ by interchanging $b_{n-1}^{(k)}$ and $b_n^{(k)}$. By $\mathcal{A}^{(0)}$ we denote the standard basis. Each time step 2 is entered the actual basis of the algorithm is some $\mathcal{A}^{(k)}$, which, moreover, fulfills $a_n^{(k)*} = 0$.

We again consider the quantity $D = \prod_{i=1}^{n-1} \|b_i^*\|^{2(n-i)}$, which is decreased by each exchange step by at least a factor of $\frac{3}{4}$. When entering in step 2 with basis $\mathcal{A}^{(k)}$, we have $D \le 1$, and by Claim 5, during execution of step 2 we have $D \ge (\min_{1\le i\le n-1} \|a_i^{(k)*}\|)^{n(n-1)}$. So the while loop is performed at most $O(n^2\cdot\log(\min\|a_i^{(k)*}\|)^{-1})$ times.

Since the other steps of the algorithm also take only a finite amount of time, either the algorithm produces an infinite sequence $\mathcal{B}^{(k)}$ of bases of $\mathbf{Z}^n$ or it stops in step 4 and outputs a vector c_n. By Lemma 2b the vector c_n is an integer relation for x; hence, an infinite sequence of bases of $\mathbf{Z}^n$ is produced if $x_1, \cdots, x_{n-1}, 1$ are $\mathbf{Z}$-linearly independent.

If $(x_1, \cdots, x_{n-1}, 1)$ are $\mathbf{Z}$-linearly dependent, the straight line $x\,\mathbf{R}$ cannot be approximated arbitrarily closely with bases of $\mathbf{Z}^n$ by (7). So by part b of the theorem, the algorithm does not output an infinite sequence of bases in this case. Hence, the algorithm produces an integer relation c_n for $x = (x_1, \cdots, x_{n-1}, 1)$ if there is one. We show $\|c_n\|^2 \le 2^{n-2}\cdot\lambda(x)^2$.

The vector c_n is defined by $[c_1, \cdots, c_n] = ([b_1, \cdots, b_n]^t)^{-1}$. Here $\{b_1, \cdots, b_n\}$ is a basis obtained from some $\mathcal{B}^{(k)}$ by interchanging $b_{n-1}^{(k)}$ and $b_n^{(k)}$. Since $c_n\,\mathbf{R}^\perp = b_n^*\,\mathbf{R}^\perp = \ll x, b_1, \cdots, b_{n-1} \gg$ and $< c_n, b_n^* > = < c_n, b_n > = 1$, we have $c_n = \pm b_n^* \cdot \|b_n^*\|$ and, therefore,

$$\|c_n\| = \|b_n^*\|^{-1}. \tag{15}$$

Now, let $m \in \mathbf{Z}^n$ be an integer relation for x such that $\|m\| = \lambda(x)$. Let i_0 be the minimal i such that $< m, b_i^{(k)} > \neq 0$. Then $| < m, b_i^{(k)} > | = | < m, b_i^{(k)*} > | \ge 1$; thus, $\|m\| \ge \|b_{i_0}^{(k)*}\|^{-1}$. We obtain from (15) and (13)

$$\begin{aligned}\|c_n\|^2 = \|b_n^*\|^{-2} = \|b_{n-1}^{(k)*}\|^{-2} &\le 2^{n-2}\cdot\|b_{i_0}^{(k)*}\|^{-2} \\ &\le 2^{n-2}\cdot\|m\|^2 = 2^{n-2}\cdot\lambda(x)^2.\end{aligned} \tag{16}$$

This finishes the proof of part (a) of our theorem. □

4.3. Proof of Theorem 7(c). The proof of part (c) of the theorem is the main contribution of this paper. We have to show that the first basis vector $b_1^{(k)} := (p_1, \cdots, p_{n-1}, q)$ of each basis $\mathcal{B}^{(k)}$ fulfills

$$\max_{1\le u\le b-1} \left| x_i - \frac{p_i}{q} \right| \le \frac{\|x\|\cdot 2^{(n+2)/4}}{|q|^{1+1/(2n(n-1))}}.$$

To this end, we first show that

$$\max_{1\leq i\leq n-1} |x_i - \frac{p_i}{q}| \leq \frac{\|b_1^{(k)*}\| \cdot \|x\|}{|q|}. \tag{17}$$

Proof of (17). Since $\|b_1^{(k)*}\|$ is the distance between $b_1^{(k)}$ and $x\,\mathbf{R}$, we have

$$\max_{1\leq i\leq n-1} \operatorname{dist}((p_i, q), (x_1, 1)\,\mathbf{R}) \leq \|b_1^{(k)*}\|. \tag{18}$$

Moreover, we have

$$\begin{aligned} \operatorname{dist}((p_i, q), (x_i, 1))\,\mathbf{R}^2 &= \|(p_i, q) - \frac{<(x_i, 1), (p_i, q)>}{x_i^2+1} \cdot (x_i, 1)\|^2 \\ &= \frac{(p_i - x_i q)^2}{x_i^2+1}. \end{aligned} \tag{19}$$

Equations (18) and (19) imply (17).

It remains to show $\|b_1^{(k)*}\| \leq 2^{(n+2)/4} \cdot |q|^{-1/(2n(n-1))}$. This will be done by showing

$$\|b_1^{(k)*}\| \leq 2^{(n+2)/4} \cdot \|b_1^{(k)}\|^{-1/(2n(n-1))},$$

which is equivalent to Proposition 10.

PROPOSITION 10. *The first vector $b_1^{(k)}$ of each output basis of the generalized cfa fulfills*

$$\|b_1^{(k)}\| \leq 2^{n+1} \cdot 2^{n(n-1)(n-2)/2} \cdot \|b_1^{(k)*}\|^{-2n(n-1)}. \tag{20}$$

The proof of Proposition 10 fills the rest of this section. We must bound $\|b_1^{(k)}\|^2 = \|b_1^{(k)*}\|^2 + |\mu_{1,0}^{(k)}|^2 \cdot \|x\|^2$ in terms of $\|b_1^{(k)*}\|$. We outline how to bound $|\mu_{1,0}^{(k)}| \cdot \|x\|$, the length of the component of $b_1^{(k)}$ parallel to $x\,\mathbf{R}$.

We shall bound $\mu^{(k)} := \max_{1\leq i\leq n} |\mu_{i,0}^{(k)}| \cdot \|x\|$. This will be done by induction on k, the number of exchange steps $n-1 \leftrightarrow n$. The induction will bound simultaneously $\mu^{(k)}$and $\nabla^{(k)}$, the norm of a linear map $f_{\mathcal{B}^{(k)}} : x\,\mathbf{R}^\perp \to \mathbf{R}$.

For any basis $\mathcal{B} = \{b_1, \cdots, b_n\}$ of the algorithm (not only output bases) we define $f_\mathcal{B} : x\,\mathbf{R}^\perp \to \mathbf{R}$ by defining $f_\mathcal{B}(\pi b_i) := \mu_{i,0} \cdot \|x\|$ for $1 \leq i \leq n-1$. The map $f_\mathcal{B}$ depends only on $\ll b_1, \cdots, b_{n-1} \gg$; we have

$$\left\{ y + f_\mathcal{B}(y) \cdot \frac{x}{\|x\|} : y \in x\,\mathbf{R}^\perp \right\} = \ll b_1, \cdots, b_{n-1} \gg .$$

For $b \in \ll b_1, \cdots, b_n \gg$ the length of the component of b parallel to $x\,\mathbf{R}$ is $\|f_\mathcal{B}(\pi b)\|$. The length of this component is bounded by $\|\pi b\| \cdot \nabla$, where ∇ is the norm of $f_\mathcal{B}$.

We have already observed that exchange steps $n-1 \leftrightarrow n$ interchange b_{n-1} and b_n without size reduction. Hence, they do not change $\mu := \max_{1\leq i\leq n} |\mu_{i,0}| \cdot \|x\|$. All the other basis transformations performed by the algorithm do not change $\ll b_1, \cdots, b_{n-1} \gg$. Hence, they leave $f_\mathcal{B}$ and thus ∇ unchanged. These observations will enable us to bound by "parallel induction" $\mu^{(k)}$ and $\nabla^{(k)}$, the values of μ and ∇ for the bases $\mathcal{B}^{(k)}$ (Lemma 11).

We now present the preceding sketched proof completely and first recall what is the norm of a linear map. Let $E \subseteq \mathbf{R}^n$ be a linear subspace, and let $f : E \to \mathbf{R}$ be a linear map. Then the norm$\|f\|$ of f is defined by $\|f\| := \sup_{y \in E, \|y\|=1} |f(y)|$. If $\{o_1, \cdots, o_r\}$ is an orthonormal basis of E such that $f(o_i) = \tau_i$ for all $i \in \{1, \cdots, r\}$, then by Riesz's lemma ([32], p. 43) we know $\|f\| = \|(\tau_1, \cdots, \tau_r)\|$. The vector $(\tau_1, \cdots, \tau_r)$ is called the *representing vector* of f with respect to $\{o_1, \cdots, o_r\}$.

Now we turn to the previously mentioned maps $f_{\mathcal{B}}$: Every basis $\mathcal{B}^{(k)}$output by our algorithm fulfills $b_n^* = 0$, since otherwise the algorithm would have stopped previously. As the reader may verify, the representing vector of $f_{\mathcal{B}}$ with respect to $\{b_1^*/\|b_1^*\|, \cdots, b_{n-1}^*/\| b_{n-1}^*\|\}$ is $V \cdot N^t \cdot D$, where $V = \|x\| \cdot (\mu_{1,0}, \cdots, \mu_{n-1,0})$, $D = (\delta_{ij}/\|b_i^*\|)_{1 \le i,j \le n-1}$, and $N = \left((\mu_{i,j})_{1 \le i,j \le n-1}\right)^{-1}$. Here δ_{ij} is the Kronecker delta. This implies

$$\|f_{\mathcal{B}}\| = \|V \cdot N^t \cdot D\|. \tag{21}$$

As in the proof of part (a), we shall also denote in the sequel by $\mathcal{A}^{(k)} = \{a_1^{(k)}, \cdots, a_n^{(k)}\}$ the basis of the algorithm After the kth exchange step $n-1 \leftrightarrow n$, so $\mathcal{A}^{(k)} = \{b_1^{(k)}, \cdots, b_{n-2}^{(k)}, b_n^{(k)}, b_{n-1}^{(k)}\}$ for $k \ge 1$. By $\mathcal{A}^{(0)}$ we denote the standard basis consisting of the unit vectors $e_1, \cdots, e_n$.

Then the value of μ is the same for $\mathcal{B}^{(k)}$and $\mathcal{A}^{(k)}$. Moreover, the maps $f_{\mathcal{A}^{(k-1)}}$ and $f_{\mathcal{B}^{(k)}}$, and thus their norms, are equal. We shall bound $\mu^{(k)}$ by looking at $\mathcal{B}^{(k)}$ and $\nabla^{(k)}$ by looking at $f_{\mathcal{A}^{(k-1)}}$. Figure 4.1 illustrates the situation.

Exchange step $n-1 \leftrightarrow n$ no.

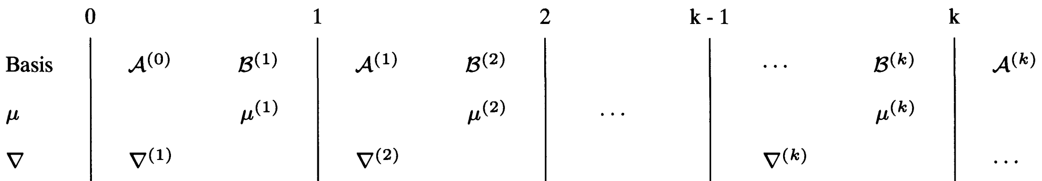


FIG. 4.1. *Exchange steps $n-1 \leftrightarrow n$ for Theorem* 7(c).

LEMMA 11. *The following inequalities hold:*

$$\nabla^{(1)} \le \prod_{i=1}^{n-1} \|e_i^*\|^{-1}; \tag{22}$$

$$\mu^{(1)} \le (n+1) \cdot \prod_{i=1}^{n-1} \|e_i^*\|^{-1}; \tag{23}$$

$$\nabla^{(k)} \le \nabla^{(k-1)} + \frac{1}{\|a_{n-1}^{(k-1)*}\|} \cdot (n-1)^{1/2} \cdot (1.5)^{n-2} \cdot \mu^{(k-1)} \quad \textit{for all } k \ge 2; \tag{24}$$

$$\mu^{(k)} \le \mu^{(k-1)} + \nabla^{(k)} \cdot 2^{n-1} \cdot \|b_{n-1}^{(k-1)*}\| \quad \textit{for all } k \ge 2. \tag{25}$$

Proof of Lemma 11.

Proof of (22). For all $y \in x\mathbf{R}^{\perp}$ we have $f_{\mathcal{A}^{(0)}}(y) \cdot x/\|x\| + y \in \ll e_1, \cdots, e_{n-1} \gg$; hence $f_{\mathcal{A}^{(0)}}(y) \cdot e_n/\|x\| + y \in \ll e_1, \cdots, e_{n-1} \gg$. This implies $\|y\| \geq \|f_{\mathcal{A}^{(0)}}(y) \cdot (e_n/\|x\|)\| = |f_{\mathcal{A}^{(0)}}(y)|/\|x\|$, so we have proved $\nabla^{(1)} \leq \|x\|$.

The claim follows from $1 = \det(L(x, e_1, \cdots, e_{n-1})) = \|x\| \cdot \prod_{i=1}^{n-1} \|e_i^*\|$.

Proof of (23). Since $\{\pi b_1^{(1)}, \cdots, \pi b_n^{(1)}\}$ is size reduced, we have $\|\pi b_j^{(1)}\| \leq (n-1)^{1/2}$ for all $j \in \{1, \cdots, n\}$. Since $b_j^{(1)} \in \ll e_1, \cdots, e_{n-1} \gg$ for these j, this implies

$$\tag{26} \begin{aligned} \max_{1 \leq i \leq n-1} |\mu_{j,0}^{(1)}| \cdot \|x\| &\leq \|\pi b_j^{(1)}\| \cdot \nabla^{(1)} \\ &\leq (n-1)^{1/2} \cdot \prod_{i=1}^{n-1} \|e_i^*\|^{-1}, \end{aligned}$$

which follows from (22).

It remains to bound $|\mu_{n,0}^{(1)}| \cdot \|x\|$. We have $b_n^{(1)} = e_n + v$ for some $v \in \ll e_1, \cdots, e_{n-1} \gg$. The bounds $\|\pi b_n^{(1)}\| \leq (n-1)^{1/2}$ and $\|\pi e_n\| \leq 1$ imply $\|\pi v\| \leq 1 + (n-1)^{1/2}$, and thus $| < v, x/\|x\| > | \leq (1 + (n-1)^{1/2}) \cdot \nabla^{(0)}$. So we have

$$\tag{27} \begin{aligned} |\mu_{n,0}^{(1)}| \cdot \|x\| &\leq | < e_n, \frac{x}{\|x\|} > | + | < v, \frac{x}{\|x\|} > | \\ &\leq 1 + (1 + (n-1)^{1/2}) \cdot \nabla^{(0)} \\ &\leq (n+1)^{1/2} \cdot \prod_{i=1}^{n-1} \|e_i^*\|^{-1}. \end{aligned}$$

Equation (23) follows from (26) and (27).

Proof of (24). Let $g = (g_1, \cdots, g_{n-1})$ be the representing vector of $f_{\mathcal{B}^{(k-1)}}$ with respect to $\left\{b_1^{(k-1)*}/\|b_1^{(k-1)*}\|, \cdots, b_{n-1}^{(k-1)*}/\|b_{n-1}^{(k-1)*}\|\right\}$. Let $\hat{g} = (\hat{g}_1, \cdots, \hat{g}_{n-1})$ be the representing vector of $f_{\mathcal{A}^{(k-1)}}$ with respect to $\left\{a_1^{(k-1)*}/\|a_1^{(k-1)*}\|, \cdots, a_{n-1}^{(k-1)*}/\|a_{n-1}^{(k-1)*}\|\right\}$. Then $\nabla^{(k-1)} = \|g\|$ and $\nabla^{(k)} = \|\hat{g}\|$. Since

$$(a_1^{(k-1)}, \cdots, a_{n-1}^{(k-1)}) = (b_1^{(k-1)}, \cdots, b_{n-2}^{(k-1)}, b_n^{(k-1)}),$$

we have $g = v \cdot N^t \cdot D$ and $\hat{g} = \hat{V} \cdot \hat{N}^t \cdot \hat{d}$, with the following notations:

$$\begin{aligned} V &= \|x\| \cdot (\mu_{1,0}^{(k-1)}, \cdots, \mu_{n-1,0}^{(k-1)}), \\ \hat{V} &= \|x\| \cdot (\mu_{1,0}^{(k-1)}, \cdots, \mu_{n-2,0}^{(k-1)}, \mu_{n,0}^{(k-1)}), \\ N &= \left((\mu_{i,j}^{(k-1)})_{1 \leq i,j \leq n-1}\right)^{-1}, \\ \hat{N} &= \left((\hat{\mu}_{i,j}^{(k-1)})_{1 \leq i,j \leq n-1}\right)^{-1} \quad \text{where } \hat{\mu}_{i,j} = \begin{cases} \mu_{i,j}, & \text{if } i \neq n-1 \\ \mu_{n,j}, & \text{if } i = n-1 \end{cases}, \\ D &= (d_{i,j})_{1 \leq i,j \leq n-1} = (\delta_{ij}/\|b_i^{(k-1)*}\|)_{1 \leq i,j \leq n-1}, \\ \hat{D} &= (\hat{d}_{i,j})_{1 \leq i,j \leq n-1} \quad \text{where } \hat{d}_{i,j} = \begin{cases} d_{i,j}, & \text{if } i \neq n-1 \\ \delta_{n-1,j}/\|a_{n-1}^{(k-1)*}\|, & \text{if } i = n-1 \end{cases}. \end{aligned}$$

The matrices D and $\hat{D}$ are diagonal matrices, and N and $\hat{N}$ are lower triangular matrices. One easily checks $g_i = \hat{g}_i$ for all $i \in \{1, \cdots, n-2\}$. This implies

$$\tag{28} \|\hat{g}_1, \cdots, \hat{g}_{n-2}\| \leq \|g\| \leq \nabla^{(k-1)}.$$

We shall now bound $|\hat{g}_{n-1}|$. To this end, we need the following general fact, which is proven readily by induction on n.

Fact. Let $M = (m_{i.j})_{1\le i,j\le n-1}$ be a lower triangular matrix such that $m_{i,i} = 1$ and $|m_{i,j}| \le m$ (say) for all $1 \le i \le n-1$ and $j < i$. Let $T = (t_{i,j})_{1\le i,j\le n-1}$ be the inverse matrix of M. Then $|t_{i,j}| \le (1+m)^{i-j}$ for all $i, j \in \{1, \cdots, n-1\}$.

Application of this fact on $(\hat{\mu}_{i,j})$ shows that each entry of the matrix $\hat{N}$ is of absolute value at most $(1.5)^{n-2}$. With this observation one checks

$$|\hat{g}_{n-1}| \le \frac{1}{\|a_{n-1}^{(k-1)*}\|} \cdot (n-1)^{1/2} \cdot (1.5)^{n-2} \cdot \mu^{(k-1)}. \tag{29}$$

Since $\nabla^{(k)} = \|\hat{g}\|$, the inequalities (28) and (29) yield the desired bound.

Proof of (25). This proof is similar, but not the same as the proof of (23). We shall use the following inequalities:

$$\|\pi b_l^{(k)}\|^2 \le 2^{n-1} \cdot \|b_{n-1}^{(k)*}\|^2 \quad \text{for all } l \in \{1, \cdots, n\} \text{ and } k \ge 1, \tag{30}$$

$$\|b_{n-1}^{(k)*}\|^2 \le 2^{n-2} \cdot \|b_{n-1}^{(k-1)*}\|^2 \quad \text{for all } k \ge 2. \tag{31}$$

Inequality (30) holds, since $2^{n-1}\|b_{n-1}^{(k)*}\|^2 = \max_{1\le i\le n} 2^i \|b_i^{(k)*}\|^2$ and since $\{\pi b_1^{(k)}, \cdots, \pi b_n^{(k)}\}$ is size reduced. Inequality (31) holds, however, by virtue of the fact that $\|b_{n-1}^{(k)*}\|^2 \le \max_{1\le i\le n} \|b_i^{(k)*}\|^2 \le 2^{n-2}\|b_{n-1}^{(k-1)*}\|^2$.

To prove (25), we first bound $\max_{1\le j\le n-1} \|x\| \cdot |\mu_{j,0}|$. For those j we have $b_j^{(k)} \in \ll a_1^{(k-1)}, \cdots, a_{n-1}^{(k-1)} \gg$. Thus,

$$\begin{aligned} \|x\| \cdot |\mu_{j,0}^{(k)}| &\le \nabla^{(k)} \cdot \|\pi b_j^{(k)}\| \\ &\le \nabla^{(k)} \cdot 2^{(n-1)/2} \cdot \|b_{n-1}^{(k)*}\| \\ &\le \nabla^{(k)} \cdot 2^{n-1} \cdot \|b_{n-1}^{(k-1)*}\| \end{aligned} \tag{32}$$

by (30) and (31).

It remains to bound $\|x\| \cdot |\mu_{n,0}^{(k)}|$. We have $b_n^{(k)} = a_n^{(k-1)} + v$ for some $v \in \ll a_1^{(k-1)}, \cdots, a_{n-1}^{(k-1)} \gg$. Since $\|\pi v\| \le \|\pi b_{n-1}^{(k-1)}\| + \|b_n^{(k)}\|$, we get

$$\begin{aligned} \|\pi v\| &\le 2^{(n-1)/2} \cdot \|b_{n-1}^{(k-1)*}\| + 2^{(n-1)/2} \cdot \|b_{n-1}^{(k)*}\| \\ &\le 2^{(n-1)/2} \cdot \|b_{n-1}^{(k-1)*}\| + 2^{(2n-3)/2} \cdot \|b_{n-1}^{(k-1)*}\| \\ &\le 2^{n-1}\|b_{n-1}^{(k-1)*}\| \end{aligned} \tag{33}$$

from (30) and (31). So we have

$$\begin{aligned} \|x\| \cdot |\mu_{n,0}^{(k)}| &\le |< b_{n-1}^{(k-1)}, \frac{x}{\|x\|} >| + |< v, \frac{x}{\|x\|} >| \\ &\le \mu^{(k-1)} + \nabla^{(k)} \cdot \|\pi v\| \\ &\le \mu^{(k-1)} + \nabla^{(k)} \cdot 2^{n-1} \cdot \|b_{n-1}^{(k-1)*}\| \end{aligned} \tag{34}$$

from (33).

Inequalities (32) and (34) yield the desired bound. This completes the proof of Lemma 11. □

We now turn the bounds of Lemma 11 into noninductive bounds for $\nabla^{(k)}$ and $\mu^{(k)}$.

LEMMA 12. *For all* $k \geq 1$ *we have*

$$\nabla^{(k)} \leq \frac{1}{\prod_{i=1}^{n-1} \|e_i^*\|} \cdot \frac{\prod_{l=1}^{k-2} \|b_{n-1}^{(l)*}\|}{\prod_{l=1}^{k-1} \|a_{n-1}^{(l)*}\|} \cdot 2^{(k-1)(2n-1)} \tag{35}$$

and

$$\mu^{(k)} \leq \frac{1}{\prod_{i=1}^{n-1} \|e_i^*\|} \cdot \frac{\prod_{l=1}^{k-1} \|b_{n-1}^{(l)*}\|}{\prod_{l=1}^{k-1} \|a_{n-1}^{(l)*}\|} \cdot 2^n \cdot 2^{(k-1)(2n-1)}. \tag{36}$$

Here the empty product as usual is defined to be 1.

Proof of Lemma 11. The proof is an induction on k. For $k = 1$, equations (35) and (36) follow from (22) and (23).

For the induction step $k - 1 \leftrightarrow k$, we need

$$\frac{\|a_{n-1}^{(k-1)*}\|}{\|b_{n-1}^{(k-2)*}\|} \leq 2^{(n-4)/2} \quad \text{for all } k \geq 2, \tag{37}$$

where $\|b_{n-1}^{(0)*}\| := 1$.

Equation (37) is obvious for $k = 2$, since $\|a_{n-1}^{(r)}\| \leq \frac{1}{2}$ for all $r \geq 1$. For $k \geq 2$ it holds since by (31)

$$\frac{\|a_{n-1}^{(k-1)*}\|}{\|b_{n-1}^{(k-2)*}\|} = \frac{\|a_{n-1}^{(k-1)*}\|}{\|b_{n-1}^{(k-1)*}\|} \cdot \frac{\|b_{n-1}^{(k-1)*}\|}{\|b_{n-1}^{(k-2)*}\|} \leq \frac{1}{2} \cdot 2^{(n-2)/2} = 2^{(n-4)/2}.$$

Now we perform the induction step $k - 1 \leftrightarrow k$. For $\nabla^{(k)}$ by substituting the induction hypothesis in (24) we obtain

$$\nabla^{(k)} \leq \frac{1}{\prod \|e_i^*\|} \cdot 2^{(k-2)(2n-1)} \cdot \left(\frac{\prod_{l=1}^{k-2} \|b_{n-1}^{(l*)}\|}{\prod_{l=1}^{k-1} \|a_{n-1}^{(l)*}\|} \right) \cdot \left(\frac{\|a_{n-1}^{(k-1)*}\|}{\|b_{n-1}^{(k-2)*}\|} + t(n) \right),$$

where $t(n) = (n-1)^{1/2} \cdot (1.5)^{n-2} \cdot 2^n$. Application of (37) yields the desired bound for $\nabla^{(k)}$, since $2^{(n-4)/2} + t(n) \leq 2^{2n-1}$ for all $n \geq 2$.

For $\mu^{(k)}$, by substituting the induction hypothesis in (25) we obtain

$$\mu^{(k)} \leq \frac{1}{\prod \|e_i^*\|} \cdot 2^{(k-2)(2n-1)} \cdot 2^n \cdot \frac{\prod_{l=1}^{k-1} \|b_{n-1}^{(l*)}\|}{\prod_{l=1}^{k-1} \|a_{n-1}^{(l)*}\|} \cdot \left(\frac{\|a_{n-1}^{(k-1)*}\|}{\|b_{n-1}^{(k-2)*}\|} \cdot 2^{-2n+1} + \frac{1}{2} \right).$$

This yields the desired bound since $\|a_{n-1}^{(k-1)*}\| \le \frac{1}{2} \cdot \|b_{n-1}^{(k-1)*}\|$. $\square$

Now we are ready to complete the proof of Proposition 10 [equation (20)].

We define $D_l := \det(L(\pi b_1^{(l)}, \cdots, \pi b_{n-1}^{(l)})) = \det(L(\pi a_1^{(l-1)}, \cdots, a_{n-1}^{(l-1)}))$ for $l \ge 1$. Then we have $D_1 = \prod_{i=1}^{n-1} \|e_i^*\|$ and $\|b_{n-1}^{(l)*}\|/\|a_{n-1}^{(l)*}\| = D_l/D_{l+1}$. This enables us to rewrite inequality (36) as $\mu^{(k)} \le 2^n \cdot 2^{(k-1)(2n-1)}/D_k$. Thus, since $\|b_1^{(k)}\| \le \|b_1^{(k)*}\| + \mu^{(k)}$, we have

$$\|b_1^{(k)}\| \le \frac{2^{n+1} \cdot 2^{(k-1)(2n-1)}}{D_k}. \tag{38}$$

Since $D_1 \le 1$ and $D_l/D_{l+1} \le \frac{1}{2}$ for all $l \ge 1$ by Claim 6, we know

$$k - 1 \le \log D_k^{-1}. \tag{39}$$

Moreover, since $\{\pi b_1^{(k)}, \cdots, \pi b_{n-1}^{(k)}\}$ is a reduced lattice basis, we know from (6) that

$$\|b_1^{(k)*}\| \le 2^{(n-2)/4} \cdot D_k^{1/(n-1)}. \tag{40}$$

Applying (39) and (40) to (38) yields

$$\|b_1^{(k)}\| \le \frac{2^{n+1} \cdot 2^{n(n-1)(n-2)/4}}{\|b_1^{(k)*}\|^{2n(n-1)}}.$$

This is equation (20), so Proposition 10 is proven. We already showed that Proposition 10 implies part c of Theorem 7. $\square$

5. Remarks on the algorithm.

(A) If we apply inequality (40) of the last section to (38) without using (39), we obtain the bound

$$\|b_1^{(k)}\| \le \frac{2^{n+1} \cdot 2^{(n-1)(n-2)/4} \cdot 2^{(k-1)(2n-1)}}{\|b_1^{(k)*}\|^{n-1}},$$

and thus

$$\max_{1 \le i \le n-1} |x_i - \frac{p_i}{q}| \le \frac{\|x\| \cdot \text{const}(n) \cdot 2^{(k-1)(2n-1)}}{|q|^{1+(n-1)^{-1}}}. \tag{41}$$

Thus, the vectors $b_1^{(k)}$ produced by our algorithm fulfill the Dirichlet bound up to some constant depending on n and x and up to a latter factor increasing with k. Maybe the latter factor comes from the inductive proof technique and maybe the algorithm really meets the Dirichlet bound up to a constant depending on n (and $\|x\|$).

(B) Let us further discuss the gap between the Dirichlet bound and our bound (41). The factor $\|x\|$ is somewhat disturbing, but for diophantine approximation problems we may assume $0 \le x_i < 1$ for $i \in \{1, \cdots, n-1\}$, and thus $\|x\| \le \sqrt{n}$.

The gap of a factor const(n) is inherent in the lattice reduction technique used.

(C) The problem of diophantine approximations alone, without the intention to generalize the cfa, was investigated by Lagarias [24] in 1982. He proposed an algorithm for rational inputs, which can immediately be carried over to real ones.

The algorithm on input $x_1, \cdots, x_{n-1}$ and $Q > 0$ produces in polynomial time a diophantine approximation $(p_1, \cdots, p_{n-1}, q)$ for $x_1, \cdots, x_{n-1}$ such that $|q| \leq Q$ and

$$\max_{1 \leq i \leq n-1} \left| x_i - \frac{p_i}{q} \right| \leq \frac{2^{n/2} \cdot n}{|q|^{1+1/(n-1)}}.$$

So the Dirichlet bound is satisfied up to some constant factor. Moreover, Lagarias proved several NP-completeness results that suggest it may be hard to find approximations within the Dirichlet bound.

(D) The vector $b \in \mathbf{Z}^n$ is called *best approximation* [7] for $x_1, \cdots, x_{n-1}$, if $\operatorname{dist}(b, x\,\mathbf{R}) < \operatorname{dist}(\hat{b}, x\,\mathbf{R})$ for all $\hat{b} \in \mathbf{Z}^n$ with $\|\hat{b}\| < \|b\|$ and $\operatorname{dist}(b, x\,\mathbf{R}) \leq \operatorname{dist}(\hat{b}, x\,\mathbf{R})$ for all $\hat{b} \in \mathbf{Z}^n$ with $\|\hat{b}\| = \|b\|$. Best approximations fulfill the Dirichlet bound. It is not known how good the diophantine approximations produced by our algorithm are compared to the best approximations.

(E) We consider the performance of our algorithm for the dimensions $n = 2$ and $n = 3$.

In the case of two dimensions the algorithm is the so-called *centered cfa* and so is the Bergman–Ferguson–Forcade algorithm. This algorithm constructs only best approximations (but not all of them). The cfa constructs both all and only best approximations.

Also, in three dimensions our algorithm and the Bergman–Ferguson–Forcade algorithm are the same. Brentjes [7] showed that the algorithm (for $n = 3$) produces all best approximations (but also other vectors). The proof uses geometrical tools of $\mathbf{R}^3$.

Vallée [37] presents and analyzes algorithms to reduce low-dimensional lattice bases. These algorithms are also based on low-dimensional geometry.

(F) In part (b) of Theorem 3.1, we bound the number of elementary basis transformations performed until $\mathcal{B}^{(k)}$ is output. The bound depends on $\|b_1^{(k)*}\|$. It would be more desirable to have a time bound of the type "For each $\rho > 0$, after $f(\rho)$ elementary basis transformations a basis $\{b_1, \cdots, b_n\}$ is output such that $\max \operatorname{dist}(b_i, x\,\mathbf{R}) \leq \rho$." However, the costs of step 2 depend on how short b_{n-1}^* is when step 2 is entered. The shorter it is, the longer step 2 can take but the smaller is $\max_{1 \leq i \leq n-1} |x_i - p_i/q|$ for the first vector of the basis output in step 3.

A priori, it is not possible to bound from below the length of b_{n-1}^* on entering step 2. However, it would be interesting to show whether for "most" inputs $x_1, \cdots, x_{n-1}$ these b_{n-1}^* are "short." This would be a metrical theory for our algorithm. Such a metrical theory exists for the cfa [35].

One could vary the algorithm in order to decrease the costs of step 2. But recall that in order to obtain the result on diophantine approximation, we had to bound $\prod_{j=1}^{n-1} \|b_j^*\|$ from below by $\|b_1^*\|$ for the output bases [equation (40)].

Acknowledgment. The results of this paper are part of the author's Ph.D. thesis guided by Professor C. P. Schnorr. Thanks to him for many helpful discussions.

REFERENCES

[1] L. Babai, *On Lovász lattice reduction and the nearest lattice point problem*, Springer Lecture Notes in Computer Science, No. 182, Springer-Verlag, Berlin, New York, 1985, pp. 13–20.

[2] L. Babai, B. Just, and F. Meyer auf der Heide, *On the limits of computations with the floor function*, Inform. and Comput., 78 (1988), pp. 99–107.

[3] J. Barkley Rosser, *A generalisation of the Euclidean algorithm to several dimensions*, Proc. Nat. Acad. Sci., 27 (1941), pp. 309–311.

[4] G. Bergman, *Notes on Ferguson and Forcade's generalized Euclidean algorithm*, preprint, Department of Mathematics, University of California, Berkeley, CA, 1980.

[5] L. Bernstein, *The Jacobi–Perron Algorithm*, Springer Lecture Notes in Mathematics, No. 207, Springer-Verlag, Berlin, New York, 1971.

[6] E. Borel, *Contribution à l'analyse arithmétique du continu*, J. Math. Pures Appl. (9), 5 (1903), pp. 329–397.

[7] A. J. Brentjes, *Multi-dimensional Continued Fraction Algorithms*, Mathematics Centre Tracts, No. 155, Amsterdam, 1982.

[8] V. Brun, *En generalisation av kjedebroken I+II*, Skr. Vid. Selsk. Kristiana, Mat. Nat. Kl., 6 (1919), pp. 1–29, and 6 (1920), pp. 1–24.

[9] J. W. S. Cassels, *An Introduction to the Geometry of Numbers*, Springer-Verlag, Berlin, New York, 1959.

[10] ———, *An Introduction to Diophantine Approximation*, Cambridge University Press, London, 1957.

[11] H. Ferguson, *A short proof of the existence of vector Euclidean algorithms*, Proc. Amer. Math. Soc., 97 (1986), pp. 8–10.

[12] ———, *A non-inductive $GL(n,\mathbf{Z})$-algorithm that constructs integral linear relations for n $\mathbf{Z}$-linearly dependent real numbers*, J. Algorithms, 8 (1987), pp. 131–145.

[13] H. Ferguson and R. Forcade, *Generalization of the Euclidean algorithm for real numbers to all dimensions higher than two*, Bull. Amer. Math. Soc., 1 (1979), pp. 912–914.

[14] ———, *Multidimensional Euclidean algorithms*, J. Reine Angew. Math., 334 (1982), pp. 171–181.

[15] C. F. Gauss, *Disquisitiones Arithmeticae*, (German translation) Springer, Berlin, 1889.

[16] R. Güting, *Zur Verallgemeinerung des Kettenbruchalgorithmus I+II*, J. Reine Angew. Math., 278/279 (1975), pp. 165–173; and 281 (1976), pp. 184–198.

[17] J. Hastad, B. Just, J. Lagarias, and C. P. Schnorr, *Polynomial time algorithms for finding integer relations among real numbers*, SIAM J. Comput. 18 (1989), pp. 859–881. Preliminary version in Springer Lecture Notes in Computer Science, No. 210, Springer-Verlag, Berlin, New York, 1986, pp. 105–118.

[18] B. Helfrich, *Algorithms to construct Minkowski reduced and Hermite reduced lattice bases*, Theoret. Comput. Sci., 41 (1985), pp. 125–139.

[19] Ch. Hermite, *Deuxième lettre á Jacobi*, Oeuvres de Hermite I, Gauthier-Villars Paris, 1905, pp. 122–135.

[20] C. G. J. Jacobi, *Allgemeine Theorie der kettenbruchähnlichen Algorithmen*, J. Reine Angew. Math., 69 (1868), pp. 29–64.

[21] R. Kannan, *Improved algorithms for integer programming and related problems*, in Proc. 15th ACM Symposium on the Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 193–206.

[22] A. Y. Khintchine, *Continued Fractions*, P. Noordhoff, Groningen, the Netherlands, 1963.

[23] A. Korkine and G. Zolotareff, *Sur les formes quadratiques*, Math. Ann., 6 (1873), pp. 366–398.

[24] J. Lagarias, *Computational complexity of simultaneous diophantine approximation problems*, in 23rd Annual Symposium on the Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1982, pp. 32–39.

[25] S. Landau, *Factoring polynomials over algebraic number fields*, SIAM J. Comput., 14 (1985), pp. 184–195.

[26] J. Lagarias and A. M. Odlyzko, *Solving low density subset sum problems*, in Proc. 24th Annual Symposium on the Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1983, pp. 1–10.

[27] A. K. Lenstra, H. W. Lenstra, and L. Lovász, *Factoring polynomials with rational coefficients*, Math. Ann., 261 (1982), pp. 513–534.

[28] H. Minkowski, *Über die positiven quadratischen Formen und über kettenbruchähnliche Algorithmen*, J. Reine Angew. Math., 107 (1891), pp. 278–297.

[29] R. E. A. C. Payley and H. D. Ursell, *Continued fractions in several dimensions*, Proc. Cambridge Philos. Soc., 26 (1930), pp. 127–144.

[30] O. Perron, *Grundlagen für eine Theorie des Jacobischen Kettenbruchalgorithmus*, Math. Ann., 64 (1907), pp. 1–76.

[31] H. Poincaré, *Sur une généralisation des fractions continues*, C. R. Acad. Sci. Paris, 99 (1884), pp. 1014–1016.

[32] M. Reed and B. Simon, *Functional Analysis*, Vol. 1, Academic Press, Orlando, FL, 1980.

[33] C. P. SCHNORR, *A hierarchy of polynomial time lattice basis reduction algorithms*, Theoret. Comput. Sci., 53 (1987), pp. 201–224.

[34] A. SCHÖNHAGE, *Factorization of univariate integer polynomials by diophantine approximation and by an improved basis reduction algorithm*, Lecture Notes in Computer Science, No. 172, Springer-Verlag, Berlin, New York, 1984.

[35] F. SCHWEIGER, *The metrical theory of Jacobi–Perron algorithm*, Springer Lecture Notes in Mathematics, No. 334, Springer-Verlag, Berlin, New York, 1973.

[36] G. SZEKERES, *Multidimensional continued fractions*, Ann. Univ. Sci. Budapest, Eötvös Sect. Math., 13 (1970), pp. 113–140.

[37] B. VALLÉE, *Une approche géométrique de la réduction de réseaux en petite dimension*, Thèse, l'Université de Caen, France, 1986.

SIAM J. COMPUT.
Vol. 21, No. 5, pp. 927–958, October 1992

© 1992 Society for Industrial and Applied Mathematics
008

LAYING OUT GRAPHS USING QUEUES*

LENWOOD S. HEATH† AND ARNOLD L. ROSENBERG‡

Abstract. The problem of laying out the edges of a graph using queues is studied. In a k-queue layout, vertices of the graph are placed in some linear order and each edge is assigned to exactly one of the k queues so that the edges assigned to each queue obey a first-in/first-out discipline. This layout problem abstracts a design problem of fault-tolerant processor arrays, a problem of sorting with parallel queues, and a problem of scheduling parallel processors. A number of basic results about queue layouts of graphs are established, and these results are contrasted with their analogues for stack layouts of graphs (the book-embedding problem). The 1-queue graphs (they are almost leveled-planar graphs) are characterized. It is proved that the problem of recognizing 1-queue graphs is NP-complete. Queue layouts for some specific classes of graphs are given. Relationships between the queuenumber of a graph and its bandwidth and separator size are presented. An apparent tradeoff between the queuewidth and the number of queues allowed in layouts of complete binary trees is indicated.

Key words. queue layout, stack layout, book embedding, graph embedding, bandwidth, separators, NP-completeness, fault-tolerant computing, scheduling parallel processors

AMS(MOS) subject classifications. 05C99, 68Q15, 68Q25, 68R10, 94C15

1. Introduction.

1.1. The problem. We study the use of queues to compute linear layouts of graphs, in the following sense. A *k-queue layout* of an undirected graph $G = (V, E)$ has two aspects. The first aspect is a linear order of V (which we think of as being on a horizontal line). The second aspect is an assignment of each edge in E to one of k queues in such a way that the set of edges assigned to each queue obeys a first-in/first-out discipline. Think of scanning the vertices in order from left to right. When the left endpoint of an edge is encountered, the edge enters its assigned queue (at the back of the queue). When the right endpoint of an edge is encountered, the edge exits its assigned queue (and must, therefore, be at the front of the queue). If a queue is examined at any instant, the edges in the queue are in the order of their right endpoints, with the leftmost of those right endpoints belonging to edges at the head of the queue. The freedom to choose the order of V and the assignment of E so as to optimize some measure of the resulting layout constitutes the essence of the queue layout problem.

More formally, a *k-queue layout* QL of an n-vertex undirected graph $G = (V, E)$ consists of a linear order of V, denoted $\sigma = 1, \cdots, n$, and an assignment of each edge in E to exactly one of k queues, $q_1, \cdots, q_k$. Each queue q_j operates as follows. The vertices of V are scanned in left-to-right (ascending) order. When vertex i is encountered, any edges assigned to q_j that have vertex i as their right endpoint must be at the front of that queue; they are removed (dequeued). Any edges assigned to q_j that have vertex i as left endpoint are placed on the back of that queue (enqueued), in ascending order of their right endpoints. k is the *queuenumber* of the layout. The *queuenumber* of G, $\mathrm{QN}(G)$, is the smallest k such that G has a k-queue layout; G is said to be a *k-queue* graph. Let $w(i, q_j)$ be the number of edges in q_j just before vertex i is encountered. Then the *queuewidth* of q_j is $\mathrm{QW}(q_j) = \max_{i \in V} w(i, q_j)$. The *maximum queuewidth*

*Received by the editors October 15, 1990; accepted for publication (in revised form) September 11, 1991. This research was supported by National Science Foundation grants DCI-87-96236, CCR-88-12567, and CCR-90-09953.

†Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061.

‡Department of Computer Science, University of Massachusetts, Amherst, Massachusetts 01003.

of the layout is $\mathrm{QW}(QL) = \max_j \mathrm{QW}(q_j)$. The *cumulative queuewidth* of the layout is $\mathrm{CQW}(QL) = \sum_j \mathrm{QW}(q_j)$.

As an example of a 1-queue layout, consider the graph G in Fig. 1.1. A 1-queue layout of G is shown in Fig. 1.2. The linear order of V is a, f, b, e, c, d. The order in which edges pass through the single queue is

$$(a, f), (a, b), (f, b), (f, e), (b, e), (b, c), (b, d), (e, d), (c, d).$$

Note that edges having the same left endpoint enter the queue in an order determined by their right endpoints. For example, edge (a, f) must enter the queue before edge (a, b) since f is to the left of b.

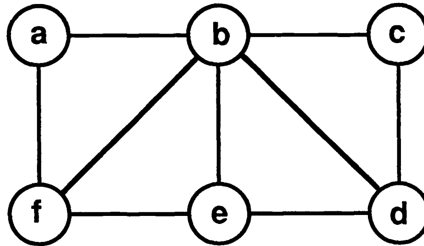


FIG. 1.1. *Example graph G.*

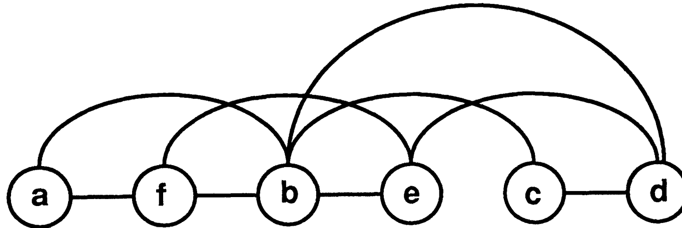


FIG. 1.2. *1-queue layout.*

Dually, a k-stack layout of graph G also has two aspects. The first aspect is again a linear order of V. The second aspect is an assignment of each edge in E to one of k stacks in such a way that the set of edges assigned to each stack obeys a last-in/first-out discipline. Unlike a queue layout, edges do not exit a stack in the same order in which they enter it.

More formally, a *k-stack layout SL* of an undirected graph consists of a linear order of V and an assignment of each edge in E to exactly one of k stacks, $s_1, \cdots, s_k$. Each stack s_j operates as follows. The vertices of V are scanned in left-to-right (ascending) order. When vertex i is encountered, any edges assigned to s_j that have vertex i as their right endpoint must be on the top of that stack; they are removed (popped). Any edges assigned to s_j that have i as left endpoint are placed on the top of the stack (pushed), in descending order of their right endpoints. k is the *stacknumber* of the layout. The *stacknumber* of G, $\mathrm{SN}(G)$, is the smallest k such that G has a k-stack layout; G is said to be a *k-stack* graph. Let $w(i, s_j)$ be the number of edges in s_j just before vertex i is encountered. Then the *stackwidth* of s_j is $\mathrm{SW}(s_j) = \max_{i \in V} w(i, s_j)$. The *maximum stackwidth* of the layout is $\mathrm{SW}(SL) = \max_j \mathrm{SW}(s_j)$. The *cumulative stackwidth* of the layout is $\mathrm{CSW}(SL) = \sum_j \mathrm{SW}(s_j)$.

As an example, Fig. 1.3 shows a 1-stack layout of the graph G in Fig. 1.1. The linear order of V is a, b, c, d, e, f. The order in which edges enter the stack is

$$(a, f), (a, b), (b, f), (b, e), (b, d), (b, c), (c, d), (d, e), (e, f).$$

The order in which edges exit the stack is

$$(a, b), (b, c), (c, d), (b, d), (d, e), (b, e), (e, f), (b, f), (a, f).$$

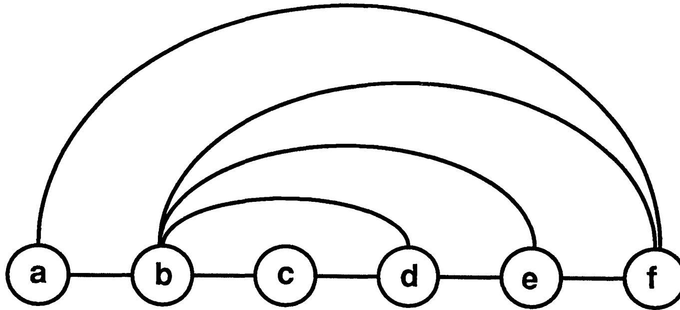


FIG. 1.3. *1-stack layout.*

The queue (respectively, stack) layout problem generalizes the problem of permuting a sequence using parallel queues (respectively, stacks) that was studied by Even and Itai [7] and Tarjan [26]. Let π be a permutation defined on $\{1, \cdots, n\}$. Define the bipartite graph G by

$$V = \{a_1, \cdots, a_n, b_1, \cdots, b_n\},$$
$$E = \{(a_i, b_i) \mid 1 \leq i \leq n\};$$

G is a perfect matching on $2n$ vertices. Then realizing π by k parallel queues (respectively, stacks) is equivalent to laying G out using k queues (respectively, stacks) when V is ordered $a_1, \cdots, a_n, b_{\pi(1)}, \cdots, b_{\pi(n)}$.

1.2. Motivation. Our study and the particular questions we focus on have a tripartite motivation.

Comparing queues and stacks. Queues and stacks are, intuitively, dual in "power" as computing mechanisms, in that queues epitomize a first-in-first-out discipline while stacks epitomize a last-in-first-out discipline. This intuition is strengthened formally when queues and stacks are used to compute fixed permutations (Tarjan [26]), largely as a consequence of the 1936 theorem of Erdös and Szekeres [6] about monotonic sequences in permutations. However, the intuition is called into doubt when queues and stacks are used as worktapes for Turing machines, because a single queue endows a Turing machine with universal computing power, whereas two stacks are needed to achieve comparable power. Here, we compare the powers of queues and stacks as devices for linearizing graphs: one "loads an edge" into the linearization device when its left end is laid out, and one "unloads" it when its right end is laid out. We find this comparison of the powers of queues and stacks to be much more complicated than the other two. To wit,

1. Stacks appear to be simpler than queues, in that the task of recognizing 1-stack graphs is computationally easy (in fact, linear time), while the analogous task for 1-queue graphs is NP-complete;
2. Queues appear to be simpler than stacks, in that, when the linearization of the vertices is preordained, the task of determining the queuenumber of the graph is computationally easy (almost linear time), while the task of determining the stacknumber is NP-complete;
3. Queues appear to be dual in power to stacks, in that a tradeoff inequality of the Erdös–Szekeres type holds for the queuenumber and stacknumber requirements of a graph when the linearization of its vertices is fixed [16];
4. Queues appear to be more powerful than stacks, in that there exist graphs whose minimum queuenumbers are exponentially smaller than their minimum stacknumbers.

These comparisons are a central theme in this paper. Note that stack layouts have been studied extensively, under the aegis of the problem of *embedding graphs in books* [2], [5], while ours is the first major study of queue layouts.

The DIOGENES design methodology. In DIOGENES [24], an array of communicating processors is implemented in a conceptual line, and some number of hardware queues and/or stacks pass over the entire line. The queues and/or stacks implement the communication links among processors in such a way that faulty processors are ignored, and all good processors are utilized. If the processors and their connections are represented by an undirected graph, then the DIOGENES layout problem is equivalent to a graph layout problem, where edges are assigned to conceptual queues and/or stacks. The variant of DIOGENES in which only stacks are used is one motivation for the studies of the book embedding problem: Bernhart and Kainen [2]; Buss and Shor [4]; Chung, Leighton, and Rosenberg [5]; Games [8]; Heath [11], [12], [13]; Heath and Istrail [14], [15]; Obrenić [21]; and Yannakakis [27]. Note that only Rosenberg [24] has considered queues before. The present research intends to investigate the same issues for queues that [5] does for stacks. In particular, we find significant instances of divergence between queue and stack layouts.

Scheduling parallel processors. Consider the following simple model of scheduling parallel computations in an architecture-independent fashion; cf. [22]. We represent the computation to be scheduled as a directed acyclic graph (dag) whose nodes represent the processes to be executed and whose arcs indicate computational dependencies: a process-node cannot be executed until all of its predecessors in the dag have been executed. Processes are queued up in a FIFO *Processor Queue* (PQ) as they become eligible for execution; each idle processor "grabs" the process at the head of the PQ. Our study focuses on the management of data in this scenario: where will the inputs to process P be when P is "grabbed" by a processor? Our solution is to have the PQ be coordinated with a *Data Manager* (DM), which itself is a collection of FIFO queues: When a process terminates, it places its "outputs" on the queues of the DM in such a way that when process P is "grabbed" by a processor, all inputs to P are at the heads of the DM queues. Our queue-based graph linearization problem idealizes this approach to the scheduling problem: The computation dag is the graph to be linearized; the linearization process implicitly specifies the loading of the PQ; the queues that control the linearization comprise the DM. In this abstract we idealize the problem even a step further by replacing the computation dag by an ordinary (undirected) graph. In subsequent work, we plan to study a more faithful version of the scheduling problem.

1.3. Results. Investigating queues at this level of generality has proved a fruitful enterprise. The harvest comprises a number of fundamental results for queue layouts as well as some surprising contrasts with stack layouts.

We summarize the highlights that are included in this paper and in the companion paper [16]. A new class of planar graphs, arched leveled-planar graphs, is shown to be a characterization of the 1-queue graphs. While 1-stack (outerplanar) graphs are easy to recognize (in fact, can be recognized in linear time), the recognition problem for 1-queue graphs is NP-complete. On the other hand, the number of queues in a *fixed-order* layout of an arbitrary graph is easily minimized in polynomial time, while the same problem for stacks is NP-complete [10]. Any 1-queue graph can be laid out with 2 stacks, and any 1-stack graph can be laid out with 2 queues [16]. An obvious generalization of these results fails: there is a class of graphs, the ternary hypercubes, that require exponentially more stacks than queues [16]. We investigate the queuenumber of some specific families of graphs and compare these to known stacknumber results. We show relationships between the queuenumber of a graph G and both the bandwidth and separator size of G. Finally, we expose an apparent tradeoff between queuenumber and queuewidth for layouts of complete binary trees.

The paper is organized as follows. Section 2 contains results on fixed-order layouts, including our polynomial-time algorithm for determining the queuenumber of such a layout. Section 3 characterizes 1-queue graphs and proves that recognizing 1-queue graphs is NP-complete; none of the later results depends on the NP-completeness proof. In §4, we investigate queue layouts for a number of familiar classes of graphs. In §5, we show relationships between the queuenumber of a graph and its bandwidth and separator size. Section 6 indicates an apparent tradeoff between queuenumber and queuewidth for complete binary trees. In the final section, we conclude with some open problems and a table comparing queuenumber and stacknumber for some specific classes of graphs.

2. Fixed-order layouts. In this section, we fix an order $\sigma = 1, 2, \cdots, n$ of V and examine the difficulty of minimizing the number of queues or stacks required to complete σ to a layout. Results include an optimal and efficient algorithm for fixed-order queue layouts. We contrast the existence of this efficient algorithm with the NP-completeness of the analogous problem for stack layouts.

We concentrate on sets of edges that are obstacles to minimizing the number of stacks or queues. A *k-rainbow* is a set of k edges

$$\{e_i = (a_i, b_i), \ 1 \leq i \leq k\}$$

such that

$$a_1 < a_2 < \cdots < a_{k-1} < a_k < b_k < b_{k-1} < \cdots < b_2 < b_1;$$

in other words, a rainbow is a *nested* matching. A *k-twist* is a set of k edges

$$\{e_i = (a_i, b_i), \ 1 \leq i \leq k\}$$

such that

$$a_1 < a_2 < \cdots < a_{k-1} < a_k < b_1 < b_2 < \cdots < b_{k-1} < b_k;$$

in other words, a twist is a fully intersecting matching.

A rainbow is an obstacle for a queue layout because no two nested edges can be assigned to the same queue.

PROPOSITION 2.1. *Assume that σ has a k-rainbow. Then every queue layout of σ uses at least k queues. There exists a stack layout of σ in which all edges of the k-rainbow are assigned to the same stack.*

A twist is an obstacle for a stack layout because no two intersecting edges can be assigned to the same stack.

PROPOSITION 2.2. *Assume that σ has a k-twist. Then every stack layout of σ uses at least k stacks. There exists a queue layout of σ in which all edges of the k-twist are assigned to the same queue.*

The largest rainbow in σ determines the smallest number of queues needed in a queue layout of σ.

THEOREM 2.3. *If σ has no rainbow of more than k edges, then there is a k-queue layout for σ. Such a layout can be found in time $O(|E| \log\log n)$.*

Proof. We describe an algorithm for assigning the edges of G to k queues, denoted $q_1, q_2, \cdots, q_k$. The algorithm uses an $(n+1)$-position array $R[0..n]$; initially, $R[0] = n+1$, and all other $R[i] = 0$. At each step of the algorithm, each array position $R[i]$, for $1 \leq i \leq n$, contains the larger of 0 and the name of the rightmost vertex of any edge that has been assigned to queue q_i to that point; the assignment $R[0] = n+1$ simplifies the algorithm, by creating the fiction that there is an edge in fictitious queue q_0 connecting fictitious vertices 0 and $n+1$. The algorithm maintains the invariant that nonzero entries in R are in strictly decreasing order: if $R[i-1] > 0$, then $R[i-1] > R[i]$. Clearly, the initial assignment to R satisfies this condition.

We actually maintain the array R in the balanced search tree data structure of Johnson [18]. The specific purpose of his data structure is to maintain a subset of a bounded set of integers $\{1, 2, \cdots, m\}$. It does so with $O(\log\log m)$ worst-case time for insertions, deletions, and accesses. As the entries in R are between 0 and $n+1$, the $O(\log n)$ time to perform a traditional binary search in R is reduced to $O(\log\log n)$.

Process the vertices in order, left to right. At each vertex s, scan the edges having s as left endpoint twice. When edge (s,t), $s < t$, is reached in the first scan, perform a binary search in R to find the queue q_i such that

$$R[i-1] > t \geq R[i].$$

Assign edge (s,t) to queue q_i. In the second scan of edges leaving s, update R to reflect the assignment of edges to queues. Clearly the algorithm maintains the conditions on R, and the edge assignment yields a queue layout.

It remains to show that, if some edge is assigned to queue q_k, then σ has a k-rainbow. Suppose (s,t) is assigned to queue q_k. Since $R[k-1] > t$ when vertex s is processed, edge (s,t) must nest inside some edge $(v, R[k-1])$ in queue q_{k-1}. Since t is assigned to queue k in the first scan, while $R[k-1]$ is updated in the second scan, the two scans at vertex s guarantee that $v < s$. By an easy induction, this observation extends to show that there are k nested edges, i.e., a k-rainbow.

The time complexity follows from the $O(\log\log n)$ search time for each edge. □

We might expect the dual result for stacks to hold; that is, if the largest twist in σ is a k-twist, then the stacknumber of σ is k, and a k-stack layout can be found in polynomial time. However, this is far from being true. For the fixed order σ, assigning edges to stacks is equivalent to coloring circle graphs [7]. While it is possible to determine the largest twist size for σ in polynomial time (Hsu [17]), minimizing the number of stacks cannot be done in polynomial time unless P = NP because coloring circle graphs is NP-complete (Garey, Johnson, Miller, and Papadimitriou [10]). To summarize, we have the following.

PROPOSITION 2.4 (see [7], [10]). *The problem of minimizing the number of stacks required by a fixed order σ is* NP-*complete.*

In this minimization sense, fixed-order queue layouts are easier than fixed-order stack layouts.

3. One-queue graphs. This section studies the class of 1-queue graphs, with the following results. Just as the 1-stack graphs admit a complete characterization as a class of planar graphs, so do the 1-queue graphs. However, whereas the 1-stack characterization is in terms of a known strengthening of the property of planarity (namely, outerplanarity), the 1-queue characterization employs a new strengthening (namely, arched-leveled planarity). Neither of these subclasses of the planar graphs includes the other. Whereas the 1-stack graphs can be recognized in linear time, we show that the recognition problem for 1-queue graphs is NP-complete. Thus, the recognition problem contrasts with the fixed-order layout problem, in that it points out a sense in which queues are more complicated than stacks.

3.1. Characterizing 1-queue graphs. Bernhart and Kainen [2] give the following characterization of 1-stack graphs.

PROPOSITION 3.1 (see [2]). *G is a* 1-*stack graph if and only if G is outerplanar.*

(An *outerplanar* graph is a planar graph having a planar embedding in which all vertices appear on a common face.) We show that the 1-queue graphs that have a particular kind of planar embedding are also planar graphs.

Consider the normal cartesian (x, y) coordinate system for the plane. For i an integer, let ℓ_i be the vertical line defined by $\ell_i = \{(i, y) \mid y \in \textbf{Reals}\}$. A graph $G = (V, E)$ is *leveled-planar* if V can be partitioned into levels $V_1, V_2, \cdots, V_m$ in such a way that

- G has a planar embedding in which all vertices of V_i are on the line ℓ_i;
- Each edge in E is embedded as a straight-line segment wholly between ℓ_i and ℓ_{i+1} for some i.

Such a planar embedding is called a *leveled-planar embedding.* Figure 3.1 shows a leveled-planar graph having 3 levels. Henceforth, we assume that a valid (but arbitrary) leveled-planar embedding is given along with a leveled-planar graph.

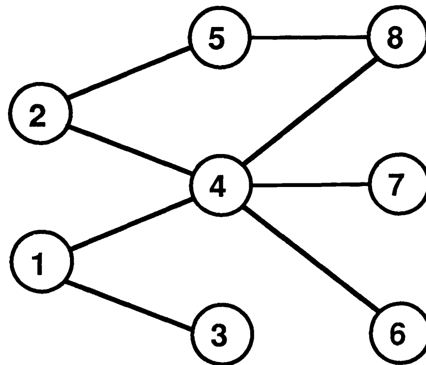


FIG. 3.1. *A leveled-planar graph.*

A leveled-planar embedding induces an order (the *induced order*) on V as follows. As i takes the values $1, 2, \cdots, m$, scan line ℓ_i from bottom to top. Label the vertices $1, 2, \cdots, n$ as they are encountered. For $1 \leq i \leq m$, let b_i be the (bottom) first vertex in level i, and let t_i be the (top) last. Let s_i be the first vertex in level i that is adjacent to some vertex in level $i + 1$, or, if there are no edges between levels i and $i + 1$, let $s_i = t_i$.

Consider augmenting G with new edges. A *level-i arch* for G is an edge connecting vertex t_i with vertex j, where $b_i \leq j \leq \min(t_i - 1, s_i)$. A leveled-planar graph G, augmented by any number of arches, can be embedded in the plane by drawing the arches around level 1; because of the leveling, the arches do not cross. See Fig. 3.2 where $(3, 5)$ and $(6, 8)$ are arches. A leveled-planar graph augmented by (zero or more) arches is called an *arched leveled-planar graph.* The edges that are not arches are called *leveled edges.* An arched leveled-planar graph that cannot be augmented with further arches or leveled edges is *maximal.* See Fig. 3.3 for an example. The above definitions for b_i, s_i, and t_i will be used throughout the paper to refer to vertices in arched leveled-planar graphs.

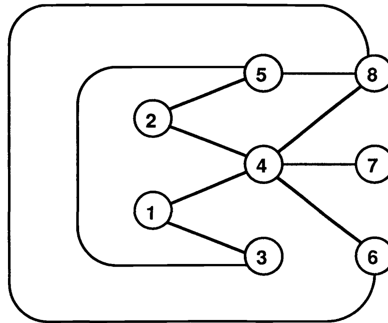


FIG. 3.2. *Drawing arches.*

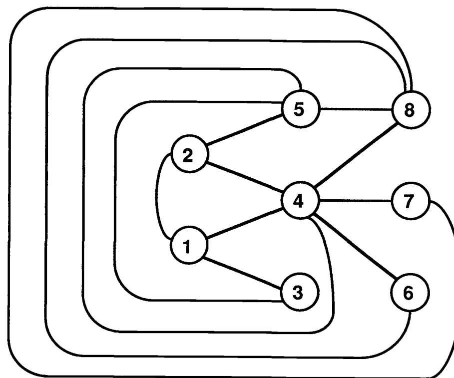


FIG. 3.3. *A maximal arched leveled-planar graph.*

We can now state the characterization of 1-queue graphs.

THEOREM 3.2. *A graph G is a 1-queue graph if and only if G is an arched leveled-planar graph.*

We develop the proof of the theorem through three lemmas.

LEMMA 3.3. *Every leveled-planar graph is a 1-queue graph. The induced order of vertices yields a 1-queue layout.*

Proof. Given a leveled-planar graph $G = (V, E)$ with m levels $V_1, V_2, \cdots, V_m$, order V in the induced order $1, \cdots, n$. We claim that this order yields a 1-queue embedding of G. It suffices to show that no two edges nest. If two edges have a vertex in common, then the edges cannot nest. So consider two edges (u_1, v_1) and (u_2, v_2) such that $u_1 < v_1$, $u_2 < v_2$, $u_1 < u_2$, and $v_1 \neq v_2$. If u_1 and u_2 are in the same level V_i, then v_1 and v_2 are in the same level V_{i+1}, and $v_1 < v_2$ because the edges do not intersect in the leveled-

planar embedding. If u_1 and u_2 are in different levels, ℓ and $m > \ell$, respectively, then v_1 and v_2 are in different levels, $\ell + 1$ and $m + 1$, respectively, and again $v_1 < v_2$. In either case, the two edges do not nest. Hence, the given layout is a 1-queue layout of G. □

LEMMA 3.4. *Every arched leveled-planar graph is a* 1*-queue graph. The induced order of vertices yields a* 1*-queue layout.*

Proof. Let $G = (V, E)$ be an arched leveled-planar graph. By the previous lemma, it suffices to show that no arch nests with another edge.

Let (u_1, t_i) and (u_2, t_j) be two arches. If $t_i = t_j$, then the arches do not nest since they have a vertex in common. If $t_i \neq t_j$, say $t_i < t_j$, then $u_1 < t_i < u_2 < t_j$, and the arches do not nest.

Now say that (u_3, v_3), $u_3 < v_3$, is a leveled edge between levels k and $k+1$. Since every arch is between two vertices on the same level, no leveled edge can nest inside an arch. Conversely, for the arch (u_1, t_i) to nest inside (u_3, v_3), we must have $u_3 < u_1 < t_i < v_3$. There are two cases:

1. If u_3 is on the same level as the arch, then $u_1 \leq u_3$, a contradiction to $u_3 < u_1$;
2. If v_3 is on the same level as the arch, then $v_3 \leq t_i$, a contradiction to $t_i < v_3$.

Thus, (u_1, t_i) and (u_3, v_3) do not nest. We conclude that we have a 1-queue layout for G. □

LEMMA 3.5. *Every* 1*-queue graph is an arched leveled-planar graph.*

Proof. Let $G = (V, E)$ be an arbitrary 1-queue graph, and let $\sigma = 1, 2, \cdots, n$ be the order of a 1-queue layout of G. It suffices to describe an arched leveled-planar embedding of G. Without loss of generality, we may assume that G is connected.

Partition V into levels, as follows. Level V_1 is the singleton $\{1\}$, so that $b_1 = s_1 = t_1 = 1$. For $i > 1$, until each vertex is placed in some level, set

- $b_i = t_{i-1} + 1$;
- t_i equal to the rightmost vertex incident to some vertex in V_{i-1};
- $V_i = \{b_i, \cdots, t_i\}$;
- s_i equal to the leftmost vertex in V_i that is adjacent to some vertex to the right of t_i; $s_i = t_i$ if $t_i = n$.

Let the resulting partition be $V_1, V_2, \cdots, V_m$. This partition breaks the sequence σ into m contiguous subsequences that end at $1 = t_1, t_2, \cdots, t_m = n$, respectively. By the construction of V_i, it is clear that no edge connects a vertex in V_i with a vertex in V_j if $|i - j| \geq 2$. Let E_ℓ be the subset of E consisting of edges that connect vertices at consecutive levels; that is,

$$E_\ell = \{(u, v) \mid \text{for some } i, u \in V_i, v \in V_{i+1}\}.$$

Construct a leveled-planar embedding of $G_\ell = (V, E_\ell)$. Place the vertices of V_i on line ℓ_i in the order $b_i, b_i + 1, \cdots, t_i$ from bottom to top. Draw the edges in E_ℓ as line segments. We must show that this is indeed a leveled-planar embedding of G_ℓ. It suffices to show that any two distinct edges, (u_1, v_1), $u_1 < v_1$, and (u_2, v_2), $u_2 < v_2$, do not cross in the embedding. Without loss of generality, say that $u_1 \leq u_2$. If $u_1 = u_2$, then the edges do not cross because they share an endpoint. So say that $u_1 < u_2$. If u_1 and u_2 are embedded on different lines, then the edges cannot intersect because all edges go between adjacent lines. If u_1 and u_2 are on the same line, say line ℓ_i, then the queuing discipline guarantees that $u_1 < u_2 < v_1 \leq v_2$ and that v_1 and v_2 are both embedded on line ℓ_{i+1}. Therefore, edges (u_1, v_1) and (u_2, v_2) do not cross.

It remains to show that $E - E_\ell$ contains only arches for G_ℓ. Let $(u_3, v_3) \in E - E_\ell$, where $u_3, v_3 \in V_i$, $u_3 < v_3$. Clearly, $u_3 \leq \min(t_i - 1, s_i)$ since otherwise there is an edge from s_i to some vertex in V_{i+1} that nests over (u_3, v_3). Since t_i is adjacent to some vertex

$x \in V_{i-1}$, we must have $v_3 = t_i$, for otherwise (x, t_i) and (u_3, v_3) nest. We conclude that (u_3, v_3) is an arch for G_ℓ. Since that edge was arbitrary, it follows that $E - E_\ell$ contains only arches, so we have constructed an arched leveled-planar embedding of G. □

Theorem 3.2 follows from Lemmas 3.4 and 3.5.

The structural result of Theorem 3.2 allows us to determine the maximum number of edges in a 1-queue graph. It is well known that a maximal outerplanar graph on n vertices contains $2n-3$ edges. A similar result is now shown for maximal arched leveled-planar graphs. These bound are useful for establishing lower bounds on queuenumber or stacknumber; cf. Proposition 4.11.

THEOREM 3.6. *Let $G = (V, E)$ be a 1-queue graph having a maximal arched leveled-planar embedding of m levels. Assume that A of the levels $V_1, \cdots, V_{m-1}$ are singletons. Then G has exactly*

$$2|V| - 1 - |V_1| - A \ \leq\ 2|V| - 3$$

edges.

Proof. Partition E into levels $E_1, \cdots, E_m$, where an edge is in level E_i if its left endpoint is in V_i. Then all level-i arches are in E_i, and E_m contains only arches. For convenience, let $t_0 = 0$.

First we count the arches in the given embedding. By maximality of the embedding,

- E_i contains $s_i - t_{i-1}$ arches if $s_i \neq t_i$ and $s_i - t_{i-1} - 1$ arches if $s_i = t_i$;
- If $b_i \neq t_i$ (i.e., if $|V_i| > 1$), then there is a leveled edge connecting $t_i - 1$ to level $i+1$, so that $s_i \neq t_i$.

Thus E_i contains $s_i - t_{i-1} - 1 = 0$ arches only when V_i is a singleton.

Now we count the leveled edges in the embedding. Each leveled edge in E_i, $1 \leq i \leq m-1$, has one endpoint among $t_i - s_i + 1$ vertices in level i and one endpoint among $t_{i+1} - t_i$ vertices in level $i+1$. By planarity, there is a bottom-to-top order on the set of leveled edges in E_i. Scanning these edges in order, the first edge connects two vertices, and each subsequent edge connects a new vertex to a previously encountered vertex (because of maximality). Thus, the number of leveled edges in E_i is

$$(t_{i+1} - t_i) + (t_i - s_i + 1) - 1 = t_{i+1} - s_i.$$

Combining the counts of the preceding two paragraphs, for $1 \leq i \leq m-1$,

$$|E_i| = \begin{cases} t_{i+1} - t_{i-1} & \text{if } |V_i| > 1, \\ t_{i+1} - t_{i-1} - 1 & \text{if } |V_i| = 1. \end{cases}$$

By analogous reasoning,

$$|E_m| = t_m - t_{m-1} - 1.$$

The cardinality of E is, therefore,

$$\begin{aligned} |E| &= \sum_{i=1}^{m} |E_i| \\ &= t_m - t_{m-1} - 1 - A + \sum_{i=1}^{m-1} (t_{i+1} - t_{i-1}) \\ &= t_m - t_{m-1} - 1 - A + t_m + t_{m-1} - t_1 \\ &= 2t_m - 1 - t_1 - A \end{aligned}$$

$$= 2|V| - 1 - |V_1| - A$$
$$\leq 2|V| - 3. \qquad \square$$

Thus the greatest number of edges that can be assigned to a single queue is $2|V|-3$. This value immediately yields a lower bound on the queuenumber of a graph.

COROLLARY 3.7. $\mathrm{QN}(G) \geq \lceil |E|/(2|V|-3) \rceil$.

3.2. Recognizing 1-queue graphs. The 1-stack graphs are exactly the outerplanar graphs and, therefore, can be recognized in linear time (Syslo and Iri [25]). In contrast, we show that the problem of recognizing 1-queue graphs is NP-complete (see Garey and Johnson [9]). Formally, the recognition problem for 1-queue graphs is the following decision problem.

ARCHED LEVELED-PLANAR.

Instance. A graph $G = (V, E)$, represented by adjacency lists.

Question. Does G have an arched leveled-planar embedding?

Rather than prove the NP-completeness of ARCHED LEVELED-PLANAR directly, we introduce a seemingly simpler decision problem.

LEVELED-PLANAR.

Instance. A graph $G = (V, E)$, represented by adjacency lists.

Question. Does G have a leveled-planar embedding?

Notice that it is not immediate that either of these problems reduces to the other. Using a rather elaborate reduction, we show that LEVELED-PLANAR is NP-complete. At the end of the section, we indicate how the reduction should be modified to show that ARCHED LEVELED-PLANAR is NP-complete.

We now present the known NP-complete problem which, via reduction, establishes the NP-completeness of LEVELED-PLANAR and, thereby, of ARCHED LEVELED-PLANAR. An instance of 3-SAT [9] is a boolean formula ϕ in conjunctive normal form such that each clause contains at most 3 literals. Let $\{v_1, v_2, \cdots, v_n\}$ be the variables of ϕ, and let $\{c_1, c_2, \cdots, c_m\}$ be the clauses. Each c_j is a set containing at most 3 literals, where each literal is either a variable v_i or the complement $\overline{v_i}$ of a variable; call a clause containing exactly k literals a *k-clause.* The *graph* of ϕ, $G(\phi) = (V(\phi), E(\phi))$ has vertex set

$$V(\phi) = \{c_j \mid 1 \leq j \leq m\} \cup \{v_i \mid 1 \leq i \leq n\}$$

and edge set $E(\phi) = E_1 \cup E_2$, where

$$E_1 = \{(c_j, v_i) \mid v_i \in c_j \text{ or } \overline{v_i} \in c_j\},$$
$$E_2 = \{(v_i, v_{i+1}) \mid 1 \leq i \leq n-1\} \cup \{(v_n, v_1)\}.$$

The edges of E_2 form a cycle called the *variable cycle.* The graph in Fig. 3.4 represents the graph of the formula having clauses $c_1 = \{v_1, \overline{v_2}, v_5\}$, $c_2 = \{\overline{v_3}, \overline{v_4}, v_5\}$, $c_3 = \{\overline{v_1}, v_2\}$, $c_4 = \{v_2, v_3, v_4\}$, and $c_5 = \{v_2, v_4, \overline{v_5}\}$.

Lichtenstein [19] shows that the following restricted version of 3-SAT is NP-complete.

PLANAR 3-SAT (P3SAT).

Instance. An instance of 3-SAT ϕ such that $G(\phi)$ is planar.

Question. Is ϕ satisfiable?

It always suffices to consider only instances such that each clause contains either two or three literals. From Lemma 1 of [19], we may assume that $G(\phi)$ has a planar embedding such that, for each v_i, all clauses containing the literal v_i are on one side of

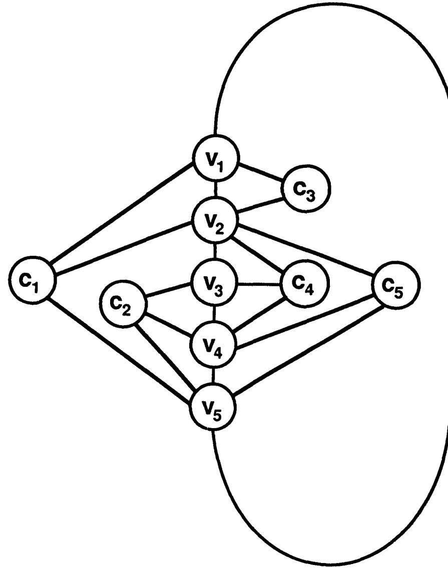


FIG. 3.4. *Example of* PLANAR 3-SAT.

the variable cycle, and all clauses containing the literal $\overline{v_i}$ are on the other side. Call this property of the planar embedding of $G(\phi)$ *consistency.* The planar embedding of Fig. 3.4 is consistent.

While LEVELED-PLANAR is as simple a recognition problem as we could formulate for queue layouts, we show that it is NP-complete in the next theorem. The proof is a long reduction from P3SAT to LEVELED-PLANAR. At the end of the subsection, the NP-completeness of ARCHED LEVELED-PLANAR is resolved by a slight modification of this reduction.

THEOREM 3.8. LEVELED-PLANAR *is* NP-*complete.*

Proof. We reduce P3SAT to LEVELED-PLANAR. As LEVELED-PLANAR is easily in NP, this suffices to prove the theorem.

Let $V = \{v_1, \cdots, v_n\}$ and $C = \{c_1, \cdots, c_m\}$ be an instance of P3SAT. Fix a planar embedding of $G(\phi)$ that is consistent. We will construct an instance H of LEVELED-PLANAR, which is a biconnected planar graph.

Here is an overview of our reduction strategy. We start with a consistent planar embedding of $G(\phi)$ in which the vertices in the variable cycle are in order on a vertical line (e.g., as in Fig. 3.4). Our strategy is to replace all vertices *and* edges in $G(\phi)$ by gadgets that have restricted leveled-planar embeddings. The graph H resulting from these replacements is "rigid" in the sense that in any leveled-planar embedding of H the *relative* levels of any two clause gadgets are fixed. A variable gadget has the flexibility of being in one of two different levels to represent *true* and *false* values of the variable. The gadget for an edge (called a *rod*) is "rigid" in the sense that the number of levels between its two ends is constant. Rods are used both to make H rigid and to transmit the truth value of a variable from the variable gadget to the gadgets of the clauses that use the variable.

As an example, consider Fig. 3.5, the rigid version of the graph from Fig. 3.4. New vertices $u_0, \cdots, u_5$ have been interspersed among $v_1, \cdots, v_5$. New vertices c_0 and c_6 have also been added as dummy clauses. All vertices and edges in Fig. 3.5 represent gadgets used in the construction of H. The construction is such that the subgraph represented

by the cycle c_0, u_0, c_6, u_5, c_0 is a fixed framework that has essentially only one leveled-planar embedding (assuming the c_0 gadget occupies earlier levels than the c_6 gadget). The subgraphs representing $u_0, \cdots, u_5$ must all appear in the same 3 adjacent levels. The subgraphs representing variables $v_1, \cdots, v_5$ have the freedom to be in 2 different positions in a leveled-planar embedding depending on their truth values. The edges that go generally left to right are replaced by rods of appropriate lengths. This is done in a way that fixes the levels occupied by the gadget of any clause. The clause gadgets have the ability to evaluate a boolean OR, in the sense that any clause gadget can successfully be placed in a leveled-planar layout if and only if at least one of its corresponding literals is assigned *true*. H has the property that it has a leveled-planar embedding if and only if ϕ has a satisfying assignment. The truth values in a truth assignment for ϕ specify a leveled-planar embedding for the variable cycle that can be extended to a leveled-planar embedding for H in the case that the assignment is a satisfying assignment.

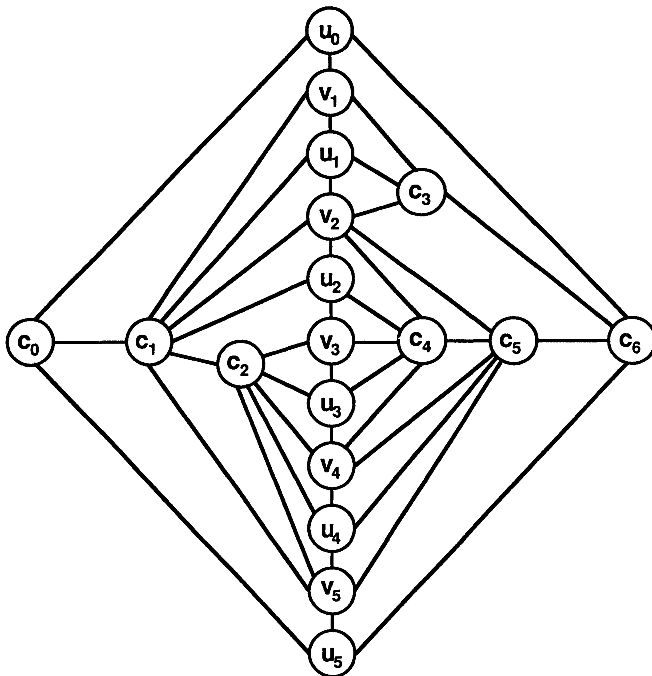


FIG. 3.5. *Rigid* PLANAR 3-SAT.

We begin the proof with some useful building blocks. Consider the copy of $K_{2,3}$ in Fig. 3.6. Suppose this copy is in a leveled-planar embedding. Then a_1 and a_2 are exactly two levels apart, and b_1, b_2, and b_3 are all on the level in between. Further, only two of b_1, b_2, and b_3 can have any additional edges incident to them. In a leveled-planar embedding, the leveling of any copy of $K_{2,3}$ is forced. If b_1, b_2, and b_3 are thought of as a single vertex, then $K_{2,3}$ is thought of as a path of length 2. In a leveled embedding, $K_{2,3}$ differs from a path of length 2 in the sense that it cannot "bend" in the middle in order to bring the ends together; its two endpoints must appear two levels apart. We think of $K_{2,3}$ as a *rigid path of length* 2. By joining $k-1$ copies of $K_{2,3}$ in the manner illustrated in Fig. 3.7 for $k = 3$, rigid paths of any length k can be obtained. In general, we call a rigid path of length k a *k-rod*. (A 1-rod is an edge.) We draw a k-rod as a thick hollow line with intermediate vertices as needed (Fig. 3.8). Note that what appears to

be a single intermediate vertex is actually two different vertices, one on the top and one on the bottom of the rod (the third vertex is inaccessible, hence ignored).

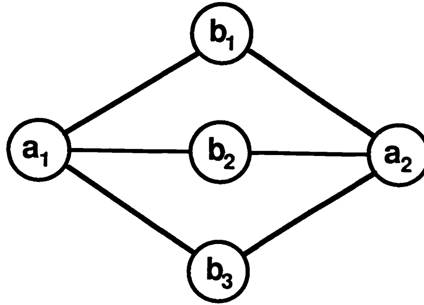


FIG. 3.6. *A 2-rod.*

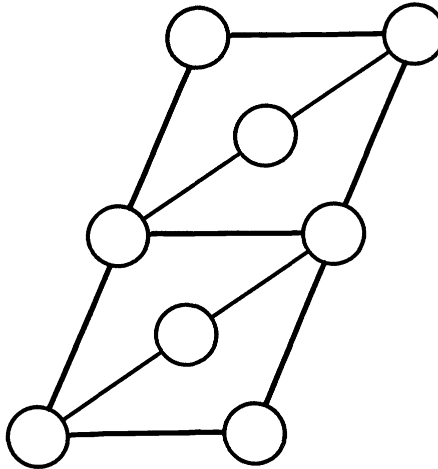


FIG. 3.7. *A 3-rod.*

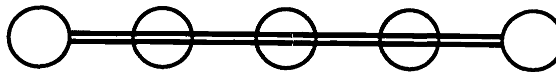


FIG. 3.8. *Representation of a k-rod, $k = 4$.*

A second building block is called a *semi-rod.* It consists of a 3-rod and a 2-rod connected by 2 edges. See Figs. 3.9 and 3.10. A semi-rod has one degree of flexibility that a 5-rod does not have: if x is in level t and y in level $t-1$, then z is either in level $t-5$ (Fig. 3.9, where the semi-rod is extended to its greatest length) or $t-3$ (Fig. 3.10, where the semi-rod is compressed); note that z is always at a lower level than y. We draw a semirod as a 5-rod with a textured interior (Fig. 3.11).

In the construction of H, some vertices will be called *fixed.* If x is a fixed vertex, we intend that, in any leveled-planar embedding of H, the level containing x is always the same (given that a particular vertex, to be specified later, is on the first level). The level in which x should appear is its *preferred level* $\mathcal{L}(x)$. If we fix the two ends of a rod, then the intermediate vertices of the rod are also fixed, with preferred levels derived in the obvious way. During the construction, we designate certain vertices x as fixed and give

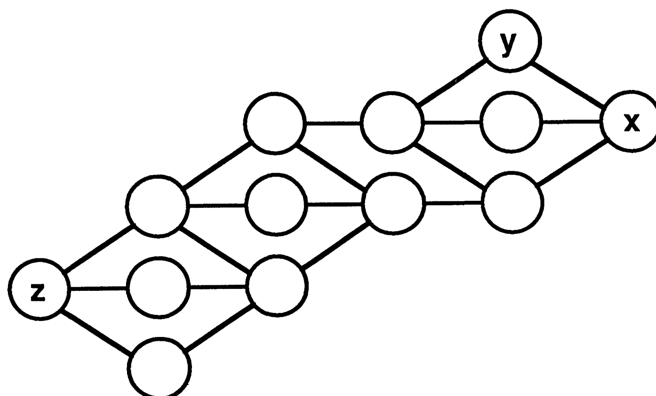


FIG. 3.9. *A semi-rod extended.*

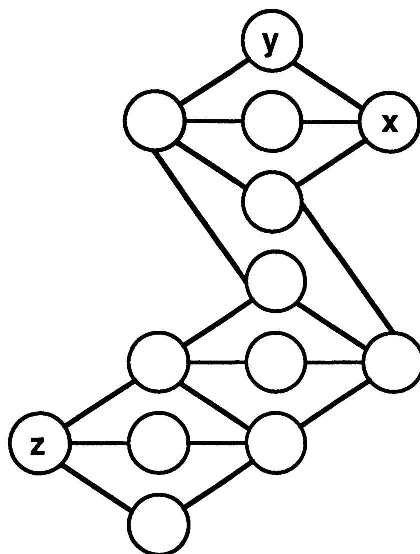


FIG. 3.10. *A semi-rod compressed.*

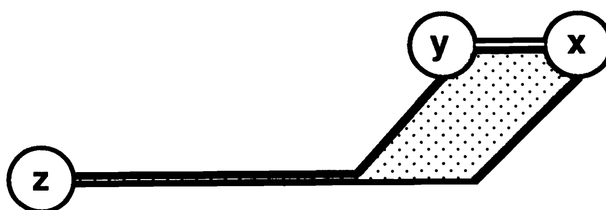


FIG. 3.11. *Representation of a semi-rod.*

a value to $\mathcal{L}(x)$. We show later that there is a leveled-planar embedding of H if and only if there is such an embedding where each fixed x indeed appears on level $\mathcal{L}(x)$.

To begin the construction for formula ϕ, we represent each variable v_i by a 2-rod $VROD[i]$ having left and right endpoints $S[i]$ and $T[i]$. (In other words, think of $S[i]$ appearing in an earlier level than $T[i]$.) Represent the edge (v_i, v_{i+1}) of $G(\phi)$, $1 \leq i \leq n-1$, by a 2-rod $EROD[i]$ having left endpoint $V[i]$ and right endpoint $W[i]$, as shown in Fig. 3.12. Partially represent the edge (v_n, v_1) by a 2-rod $EROD[0]$ connected to $VROD[1]$ and by a 2-rod $EROD[n]$ connected to $VROD[n]$ (the representation of the edge will be completed later). Call the graph constructed so far P. P may be thought of as a path of thickness 3 from $EROD[0]$ to $EROD[n]$. The vertices of each $EROD$ are fixed, and the vertices of each $VROD$ are not fixed. Note that, in any leveled-planar embedding containing P, the levels of $T[i]$ and of $W[i]$ differ by exactly one level. Further, if $W[0]$ is to the right of $V[0]$, then each $W[i]$ is to the right of $V[i]$, and vice versa. By symmetry, we may assume that any leveled-planar embedding of P has each $W[i]$ to the right of $V[i]$, and, therefore, each $\mathcal{L}(V[i]) = \mathcal{L}(W[i]) - 2$, $0 \leq i \leq n$. The intention of the construction (not yet realized) is that all $W[i]$'s appear on the same level, namely, $\Lambda = \mathcal{L}(W[0])$. For the time being, we use the level Λ as a relative reference for other $\mathcal{L}$ values. Call the property of all $W[i]$'s appearing on level Λ *line up*.

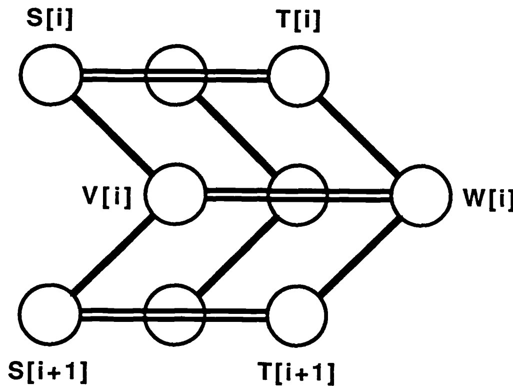


FIG. 3.12. *Representing the variable path.*

Because the embedding of $G(\phi)$ is planar, the variable cycle partitions the clause set C into two subsets, C_1 and C_2, in such a way that the clauses in C_1 nest and the clauses in C_2 nest. We will place the clauses in C_1 to the left of P and the clauses in C_2 to the right. (In Fig. 3.4, $C_1 = \{c_1, c_2\}$ and $C_2 = \{c_3, c_4, c_5\}$. Clause c_2 is nested under clause c_1, and clause c_4 is nested under clause c_5.) A clause $c_j \in C_2$ is associated with the 2 or 3 $T[i]$'s that correspond to its variables. (Similarly, a clause in C_1 is associated with 2 or 3 $S[i]$'s.) Because the embedding of $G(\phi)$ is consistent, each $T[i]$ can be associated consistently with either v_i or $\overline{v_i}$ (the corresponding $S[i]$ is associated with the complementary literal). If the $W[i]$'s line up (on level Λ), then each $T[i]$ appears either on level $\Lambda + 1$ or $\Lambda - 1$. If $T[i]$ is on level $\Lambda - 1$, then we say that $T[i]$ is *intruded* and $S[i]$ is *extruded;* otherwise (i.e., $T[i]$ is on level $\Lambda + 1$), $T[i]$ is *extruded* and $S[i]$ is *intruded.* We interpret the literal associated with each $T[i]$ (or $S[i]$) as *true* or *false*, respectively, according as whether $T[i]$ (or $S[i]$) is intruded or not.

We need gadgets for each clause in C. By mirror-image symmetry in P, we consider only clauses in C_2 and place their gadgets to the right of P. Construct the gadgets for the clauses in C_2 in any order that obeys the nesting of clauses, taking the more deeply

nested clauses earlier. (In Fig. 3.4, construct the gadget for c_4 before the gadget for c_5; the gadget for c_3 is unconstrained.) Figure 3.13 shows the gadget for a clause. The gadget contains two semirods that share an edge. $U[j]$, $X[j]$, and $Y[j]$ (among others) are fixed vertices. $X[j]$ and $Y[j]$ must appear in the same level, which is 4 levels before the level of $U[j]$. $Q[j,1]$, $Q[j,2]$, and $Q[j,3]$ are connected to the literals in the clause by rods. They may be either 3 or 5 levels before $U[j]$, depending on the truth value of the corresponding literal.

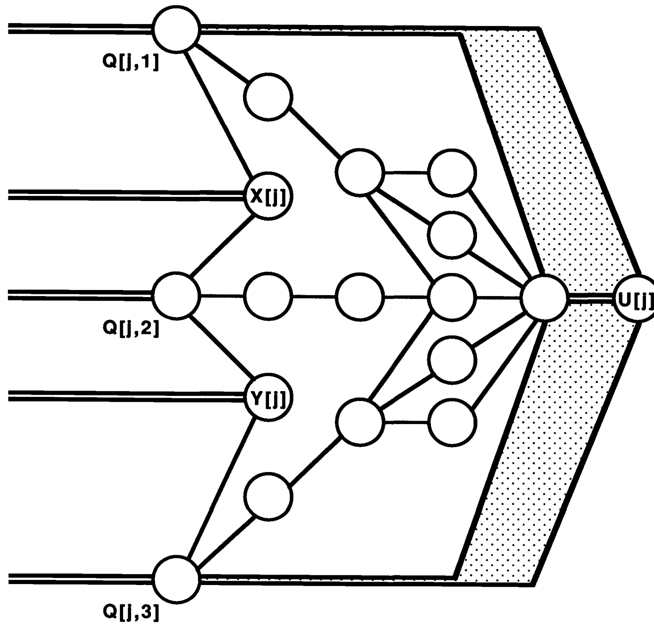


FIG. 3.13. *The gadget for a* 3*-clause.*

We first assume that $c_j \in C_2$ is a 3-clause. Let c_j be associated with $T[i_1]$, $T[i_2]$, and $T[i_3]$ in order from top to bottom. By the construction regimen, the gadgets for any clauses nested inside c_j have already been constructed. The gadget for each $c_s \in C_2$ contains a fixed vertex $U[s]$ that is visible on the right side of the gadget. If no clauses nest under c_j, then $\mathcal{L}(U[j]) = \Lambda+6$. If there are one or more clauses nested under c_j, let c_s be one that maximizes $\mathcal{L}(U[s])$. Put $\mathcal{L}(U[j]) = \mathcal{L}(U[s])+6$. Let $k = \mathcal{L}(U[j])-\Lambda-4$. Place a k-rod on each of $T[i_1], T[i_2], T[i_3]$, and connect them to $Q[j,1], Q[j,2], Q[j,3]$, as shown by the k-rods on the left in Fig. 3.13. $Q[j,a]$ is *intruded* (*extruded*) exactly when $T[i_a]$ is intruded (extruded). $X[j]$ and $Y[j]$ are fixed with $\mathcal{L}(X[j]) = \mathcal{L}(Y[j]) = \mathcal{L}(U[j]) - 4$. There will be $U[s]$'s or $W[i]$'s visible under $X[j]$ (or $Y[j]$). Connect a rod of the appropriate length from $X[j]$ (or $Y[j]$) to each visible $U[s]$ and $W[i]$. For example, between $X[j]$ and $U[s]$, connect a $(\mathcal{L}(X[j]) - \mathcal{L}(U[s]))$-rod.

In the case that c_j is a 2-clause, the gadget is the same. There are only two vertices, $T[i_1]$ and $T[i_2]$, associated with c_j. Connect $T[i_1]$ to $Q[j,1]$ and $T[i_2]$ to $Q[j,3]$ with rods as before. There will be at least one fixed vertex visible under $X[j]$, $Y[j]$, and $Q[j,2]$ (that is, some $U[s]$ or $W[i]$). Connect rods of appropriate lengths between one such

fixed vertex and $X[j]$, $Y[j]$, and $Q[j, 2]$, so that $Q[j, 2]$ is always extruded. Then c_j is represented by the gadget in the same manner as a 3-clause in which the second literal is always false. This allows us to treat every clause as though it were a 3-clause.

Once gadgets have been constructed for each clause to the right of P, *cap* the right end of H with a path around the right end, in the following sense. Let $c_s \in C_2$ have maximum level $\mathcal{L}(U[s])$. (If $C_2 = \emptyset$, by abuse of notation, take $\mathcal{L}(U[s]) = \Lambda$.) Let $k = \mathcal{L}(U[s]) - \Lambda + 3$. Place a k-rod at each of $W[0]$ and $W[n]$; identify their free ends (Fig. 3.14). Notice that two edges, one from each rod, are also identified as the edge $(X[m+1], Z[right])$. $X[m+1]$ is a fixed vertex with $\mathcal{L}(X[m+1]) = \Lambda + k - 1$. Some $U[s]$'s or $W[i]$'s will be visible from $X[m+1]$. Connect $X[m+1]$ to each of them with a rod of appropriate length. The cap around the right end may be thought of as a dummy clause containing no literals whose purpose is to provide a rod for any $U[s]$'s and $W[i]$'s that have yet to be connected to a rod.

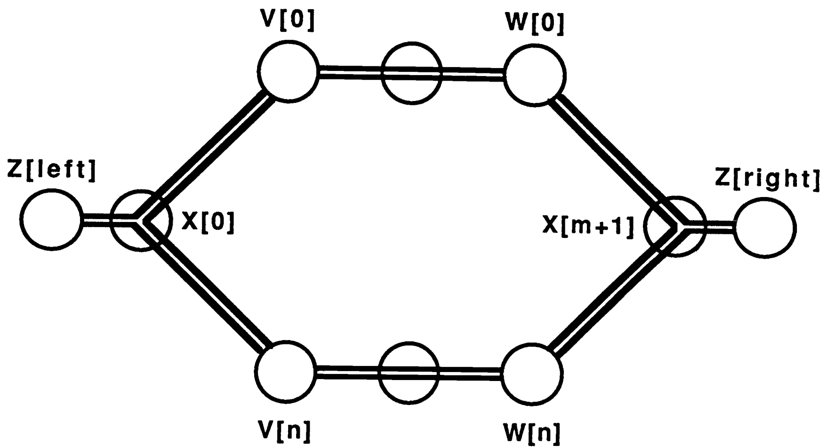


FIG. 3.14. *Capping the left and right ends.*

After the gadgets for the clauses to the left of P are constructed, cap the left end by two rods from $V[0]$ to $X[0]$ and from $V[n]$ to $X[0]$ in a manner similar to the preceding paragraph. This completes the construction of H.

So far, all the $\mathcal{L}$ values have been relative to $\Lambda = \mathcal{L}(W[0])$. Now fix $\mathcal{L}(Z[left]) = 1$, so that

$$\Lambda = \mathcal{L}(W[0]) - \mathcal{L}(Z[left]) + 1.$$

Remaining $\mathcal{L}$ values are adjusted according to the value of Λ.

Clearly, the construction can be accomplished in polynomial time. Also, H is a planar graph with a planar embedding that is essentially unique except for some inconsequential freedom in embedding the intermediate vertices in k-rods.

Every $V[i]$, $W[i]$, and $U[j]$ in H has a rod connecting it to either an $X[j]$ or a $Y[j]$. If H has a leveled-planar embedding, then it has a leveled-planar embedding in which (1) $Z[left]$ is in level 1; (2) every vertex in the capping cycle

$$Z[left], \cdots, V[0], \cdots, W[0], \cdots, Z[right], \cdots, W[n], \cdots, V[n], \cdots, Z[left]$$

is in its preferred level; (3) $W[0]$ is above $W[n]$ in level Λ. Because the capping cycle has only one leveled-planar embedding satisfying these constraints, all other vertices are

forced to be inside the capping cycle. In such an embedding, we want each fixed vertex to be in its preferred level. We show in the following two claims that this must be the case. In Claim 1, we assume that, for a particular clause $c_j \in C_2$, $U[j]$ is in level t, $X[j]$ and $Y[j]$ are in level $t-4$, the rod connected to $U[j]$ goes right, and the rods connected to $Q[j,1], X[j], Q[j,2], Y[j], Q[j,3]$ go left.

CLAIM 1. The gadget for c_j has such a leveled-planar embedding if and only if at least one of $Q[i,1], Q[i,2], Q[i,3]$ is intruded.

Proof. If $Q[j,1]$, $Q[j,2]$, and $Q[j,3]$ are all intruded, Fig. 3.13 shows such an embedding. If only $Q[j,2]$ is intruded, Fig. 3.15 shows such an embedding. If only $Q[j,1]$ (or, by symmetry, only $Q[j,3]$) is intruded, Fig. 3.16 shows such an embedding. If two of $Q[j,1]$, $Q[j,2]$, and $Q[j,3]$ are intruded, moving an appropriate $Q[j,a]$ in one of these figures easily gives an embedding. From these three figures, it is clear that there is no leveled-planar embedding of the gadget if all three vertices $Q[j,1]$, $Q[j,2]$, and $Q[j,3]$ are extruded. □

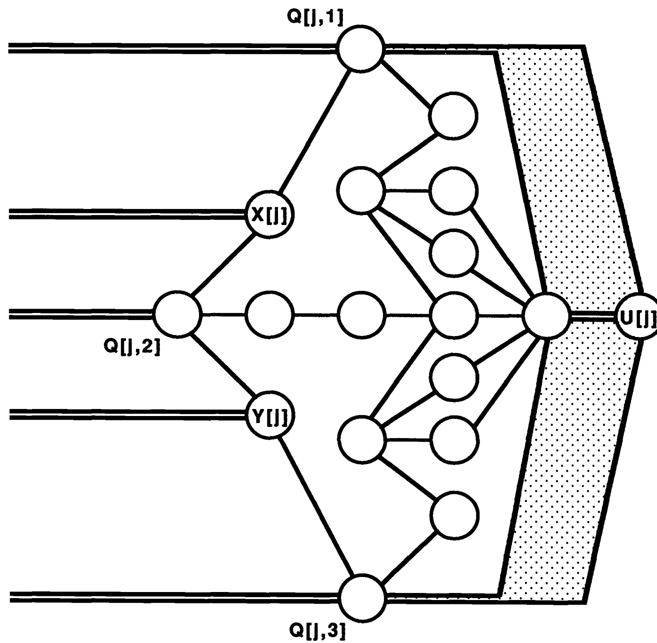


FIG. 3.15. $Q[j,2]$ *intruded.*

CLAIM 2. If H has a leveled-planar embedding such that each vertex in the capped cycle is in its preferred level, then each fixed vertex of H is in its preferred level.

Proof. Say that there is a fixed vertex not in its preferred level. If there is such a vertex in an $EROD$, let i be a smallest index for which $EROD[i]$ contains such a vertex. By left-right symmetry, it suffices to consider the case that $W[i]$ is in a level $t > \Lambda$. Because i is minimum, $t = \Lambda + 2$. $W[i]$ is connected by a rod to either an $X[j]$ or a $Y[j]$. Without loss of generality, say that the rod is to $X[j]$. The rod forces $X[j]$ to be in level $\mathcal{L}(X[j]) + 2 = \mathcal{L}(U[j]) - 2$. We claim that $U[j]$ is in a level higher than $\mathcal{L}(U[j])$.

Suppose $U[j]$ is in level $\mathcal{L}(U[j])$. The semi-rod to $Q[j,1]$ forces $Q[j,1]$ to be in level $\mathcal{L}(U[j]) - 3$. (Otherwise, $W[i]$ could not be in level $\Lambda + 2$.) The rods of $X[j]$ and $Q[j,1]$

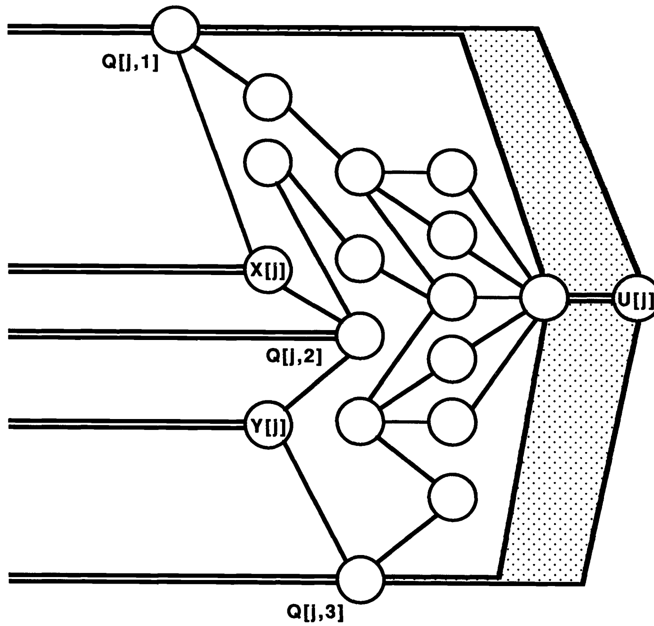


FIG. 3.16. $Q[j,1]$ *intruded.*

force $Q[j,1]$ to be above $X[j]$. Therefore, it is not possible to embed the paths from $Q[j,1]$ to $U[j]$ properly in levels.

We conclude that $U[j]$ is in level $\geq \mathcal{L}(U[j]) + 2$. The above argument repeats with the rod from $U[j]$ connecting to some $X[j']$ or $Y[j']$, shifting $U[j']$ to a level higher than $\mathcal{L}(U[j'])$. Repetition of the argument ends at $X[m+1]$, which must be in level $\mathcal{L}(X[m+1])$, not higher. This contradiction proves that each $W[i]$ is in level Λ.

A similar argument shows that any fixed vertex that is not a $W[i]$ must also be in its preferred level. □

We need to show that ϕ is satisfiable if and only if H has a leveled-planar embedding.

Assume that ϕ is satisfiable. Choose a satisfying assignment for ϕ. Embed P first. Place all fixed vertices of P on their assigned levels. If v_i is true, let whichever of $S[i]$ and $T[i]$ corresponds to the literal v_i be intruded. If v_i is false, let whichever of $S[i]$ and $T[i]$ corresponds to the literal $\overline{v_i}$ be intruded. Then each $u[j]$ has at least one intruded $Q[i,j]$ and can be level embedded by Claim 1. Thus H has a leveled-planar embedding.

Now assume that H has a leveled-planar embedding. By Claim 2, we may assume that each fixed vertex F is on level $\mathcal{L}(F)$. Let $Z[i]$ be whichever of $S[i]$ and $T[i]$ corresponds to the literal v_i. If $Z[i]$ is intruded, assign v_i the value true; otherwise, assign v_i the value false. By Claim 1, every $U[j]$ has an intruded $Q[i,j]$. Therefore, each clause c_j contains a literal that is true under this assignment. This truth assignment satisfies ϕ; that is, ϕ is satisfiable.

Thus P3SAT reduces to LEVELED-PLANAR. As P3SAT is NP-complete, we conclude that LEVELED-PLANAR is NP-complete. □

It appears that the graph H is arched leveled-planar if and only if it is leveled-planar. To be certain of this, we modify the construction slightly by adding an *arched cap* on the left and right ends of H. The cap on the right end is shown in Fig. 3.17. The rightmost

edge of the right cap and the leftmost edge of the left cap must be arches, and no other edges may be arches. With this change to H, H is an arched leveled-planar graph if and only if ϕ is satisfiable. This proves the following corollary.

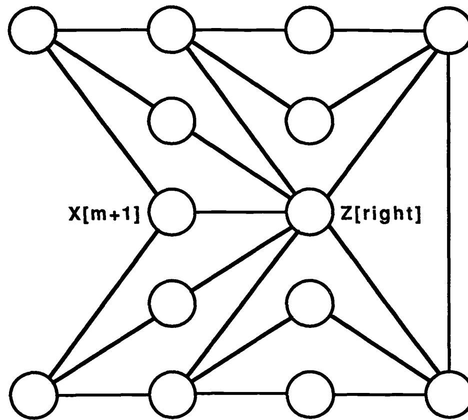


FIG. 3.17. *An arched cap.*

COROLLARY 3.9. ARCHED LEVELED-PLANAR *is* NP-*complete. Thus, the problem of recognizing* 1-*queue graphs is* NP-*complete.*

4. Layouts for specific graphs. In this section, we present queue layouts having small queuenumber for a variety of specific families of graphs. We make contrasts with the corresponding stack layouts. The intuition that an easily leveled graph has a good queue layout is supported by most of these families. Some details are left to the reader.

4.1. Trees and meshes. We begin with trees and meshes, two natural leveled-planar families of graphs.

A *tree* T is a connected graph that has no cycles. Choose an arbitrary vertex r to be the *root* of T. Each vertex in T has a well-defined *depth*, i.e., distance from r. Let $\mathrm{DEPTH}(i)$, $i \geq 0$, consist of all vertices at depth i.

PROPOSITION 4.1. *Every tree T is a leveled-planar, hence* 1-*queue, graph. T has a* 1-*queue layout such that the first level is $\{r\}$ and the queuewidth of the layout is the cardinality of the largest* $\mathrm{DEPTH}(i)$, $i > 0$.

Proof. Lay T out breadth-first starting from the root r. The result is a 1-queue layout of the tree with the stated properties. □

An $m \times n$ *mesh* is a graph with vertices

$$\{v_{ij} \mid 1 \leq i \leq m, 1 \leq j \leq n\}$$

and edges

$$\{(v_{ij}, v_{i,j+1}) \mid 1 \leq j \leq n-1\} \cup \{(v_{ij}, v_{i+1,j}) \mid 1 \leq i \leq m-1\}.$$

PROPOSITION 4.2. *An $m \times n$ mesh is a leveled-planar, hence* 1-*queue, graph. There is a* 1-*queue layout QL of the mesh having queuewidth* $\mathrm{QW}(QL) \leq \min\{m, n\}$.

Proof. An $m \times n$ mesh has a natural embedding in the plane with vertices in m rows and n columns. If this embedding is rotated 45°, vertices line up on $m + n - 1$ vertical lines. The result is a leveled-planar embedding with the stated queuewidth. □

For $m > 2$, $n > 2$, an $m \times n$ mesh is not an outerplanar graph and, in fact, has stacknumber 2 [5]. Thus, the mesh provides an example of a 1-queue graph that fails to be a 1-stack graph.

4.2. Unicyclic graphs. A *unicyclic* graph is an undirected graph in which each connected component contains at most one cycle. The family of unicyclic graphs includes trees, forests, and cycles of all lengths.

PROPOSITION 4.3. *A unicyclic graph is an arched leveled-planar, hence* 1*-queue, graph. Each connected component contributes at most one arch.*

Proof. Let $G = (V, E)$ be a unicyclic graph. We may assume that G is connected. By Proposition 4.1, we need only treat the case that G contains a cycle. Let

$$C = u_1, u_2, \cdots, u_k, u_1$$

be that cycle. If k is even, level C into $\frac{k}{2} + 1$ levels

$$U_1 = \{u_1\}, U_2 = \{u_2, u_k\}, \cdots, U_i = \{u_i, u_{k-i+2}\}, \cdots, U_{(k/2)+1} = \left\{u_{(k/2)+1}\right\};$$

a leveled-planar embedding of C results. If k is odd, level C into $\frac{k+1}{2}$ levels

$$U_1 = \{u_1, u_k\}, U_2 = \{u_2, u_{k-1}\}, \cdots, U_i = \{u_i, u_{k-i+1}\}, \cdots, U_{(k+1)/2} = \left\{u_{(k+1)/2}\right\};$$

an arched leveled-planar embedding of C results, with the single arch (u_1, u_k).

Let G' be G without the edges of C. G' contains one connected component for each u_i; this connected component is a tree T_i, which we root at u_i. We convert the (arched) leveled-planar embedding of C into one for G by expanding T_i from u_i in a breadth-first manner, as prescribed in Proposition 4.1. □

4.3. X-trees. The *depth-d complete binary tree* $CBT(d)$ has vertex set

$$\{1, 2, \cdots, 2^{d+1} - 1\}$$

and edge set

$$\{(\alpha, 2\alpha), (\alpha, 2\alpha + 1) \,|\, 1 \leq \alpha \leq 2^d - 1\}.$$

The root of $CBT(d)$ is 1, and $CBT(d)$ has $d+1$ levels in the leveling starting at the root. The *depth-d X-tree* $X(d)$ is the supergraph of $CBT(d)$ that has edges added across each of the levels from left to right. See Fig. 4.1.

Every $X(d)$ is a 2-stack graph; when $d \leq 2$, $X(d)$ is a 1-stack graph [5]. In contrast, even small X-trees require two queues.

PROPOSITION 4.4. *For $d \geq 1$, $X(d)$ admits a* 2*-queue layout with queuewidths 2^d and* 1*. For $d \geq 2$, $X(d)$ is not a* 1*-queue graph.*

Proof. For the upper bound, choose the order $\sigma = 1, 2, \cdots, 2^{d+1} - 1$. The edges of $CBT(d)$ are assigned to one queue and the edges across each level are assigned to a second queue.

For the lower bound, since $X(2)$ is a subgraph of $X(d)$, $d \geq 2$, it suffices to show that $X(2)$ is not a 1-queue graph.

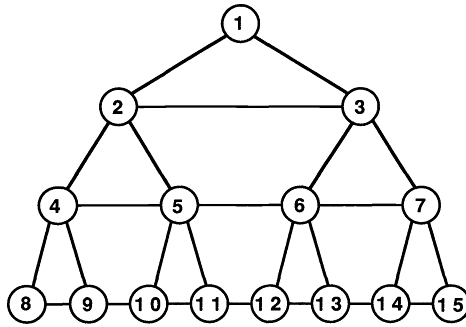


FIG. 4.1. *X-tree* $X(3)$.

We exploit an alternate means of constructing $X(2)$. Given any graph G and any edge (x, y) in G, define the operation of *hatting* (x, y) as adding a new vertex (the *peak*) z and new edges (x, z) and (y, z). Start with a cycle of length 4:

$$C = u_1, u_2, u_3, u_4, u_1.$$

Choose any three of the four edges of C. Hat each of the chosen edges. The resulting graph is isomorphic to $X(2)$.

To obtain a contradiction, suppose that $X(2)$ has a 1-queue layout. Let σ be the order of the vertices. Without loss of generality, assume that u_1 is the leftmost vertex of C in σ. Neither u_2 nor u_4 can be the rightmost vertex of C in σ, for then two edges of C would nest. By symmetry we may assume that the order of the vertices of C in σ is u_1, u_2, u_4, u_3. Three of the four edges of C must be hatted. In particular, either u_1 or u_3 has both of its incident edges hatted. By symmetry, we may assume that (u_1, u_4) and (u_1, u_2) are hatted. Let w be the peak of (u_1, u_4).

There are five possible placements of w within the order u_1, u_2, u_4, u_3. Only placement of w between u_1 and u_2 fails to yield two nested edges. But, with w between u_1 and u_2, there is no placement of the peak of (u_1, u_2) that does not yield two nested edges. This is a contradiction to σ giving a 1-queue layout of $X(2)$. □

Since $X(2)$ is outerplanar, we have the following corollary.

COROLLARY 4.5. *$X(2)$ is a 1-stack graph that is not a 1-queue graph.*

4.4. DeBruijn graphs. The *order-d deBruijn graph $DB(d)$* has vertex set

$$\{0, 1, \cdots, 2^d - 1\}$$

and edges connecting each vertex x with vertices $2x \bmod 2^d$ and $2x+1 \bmod 2^d$. See Fig. 4.2. Note that multiple edges and loops are discarded.

PROPOSITION 4.6. *$DB(d)$ admits a 2-queue layout with queuewidths 2^{d-1}. $DB(d)$, $d \geq 4$, does not admit a 1-queue layout. $DB(3)$ does admit a 1-queue layout.*

Proof. The edges of $DB(d)$ of the forms $(x, 2x)$ and $(x, 2x+1)$, $x \in \{1, 2, \cdots, 2^{d-1} - 1\}$, are the edges of a depth-$(d-1)$ complete binary tree rooted at vertex 1 and containing all vertices except 0. Similarly, the edges of the forms $(2^{d-1}+x, 2x)$ and $(2^{d-1}+x, 2x+1)$, $x \in \{0, 1, \cdots, 2^{d-1} - 2\}$, are the edges of a depth-$(d-1)$ complete binary tree rooted at vertex $2^{d-1} - 2$ and containing all vertices except $2^{d-1} - 1$. Choose the order $\sigma = 0, 1, \cdots, 2^d - 1$. Assign edges of the forms $(x, 2x)$ and $(x, 2x+1)$ to one queue and edges of the forms $(2^{d-1} + x, 2x)$ and $(2^{d-1} + x, 2x + 1)$ to a second queue. (When edges are assigned to both queues, break ties arbitrarily.)

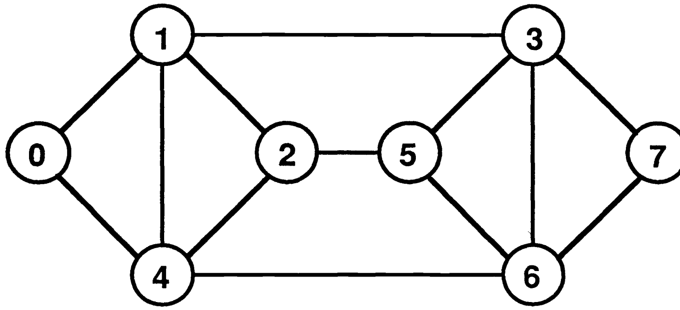


FIG. 4.2. *The deBruijn graph $DB(3)$.*

$DB(d)$, $d \geq 4$, is not planar, hence not a 1-queue graph. The order

$$\sigma = 1, 0, 2, 3, 4, 5, 7, 6$$

yields a 1-queue layout of $DB(3)$. □

4.5. Complete graphs. The *complete graph* K_n has a vertex set of size n and an edge connecting every pair of vertices.

PROPOSITION 4.7. $\mathrm{QN}(K_n) = \lfloor n/2 \rfloor$.

Proof. Every vertex order for K_n is symmetric, so fix any order $\sigma = 1, 2, \cdots, n$. The maximum size of a set of nesting edges is exactly $\lfloor n/2 \rfloor$. By Proposition 2.1 and Theorem 2.3, the result follows. □

An explicit assignment of edges of K_n to queues is easily described. In the fixed order σ, every edge (i, j) has length $|i - j|$. There are edges of every length from 1 to $n - 1$. For $i \in \{1, 2, \cdots, \lfloor n/2 \rfloor\}$, assign all edges of lengths $2i - 1$ and $2i$ to queue q_i. No two edges having the same length or having lengths differing by 1 can nest.

4.6. Complete bipartite graphs. The *complete bipartite graph* $K_{m,n}$ has $m + n$ vertices, partitioned into two sets:

$$\{a_1, a_2, \cdots, a_m\} \cup \{b_1, b_2, \cdots, b_n\};$$

its edges connect every a vertex with every b vertex.

PROPOSITION 4.8. $\mathrm{QN}(K_{m,n}) = \min(\lceil m/2 \rceil, \lceil n/2 \rceil)$.

Proof. Without loss of generality, assume that $m \leq n$. We need to show that

$$\mathrm{QN}(K_{m,n}) = \lceil m/2 \rceil.$$

Upper bound. Choose the layout order

$$\sigma = a_1, a_2, \cdots, a_{\lceil m/2 \rceil}, b_1, \cdots, b_n, a_{\lceil m/2 \rceil + 1}, \cdots, a_m.$$

Partition the edges of $K_{m,n}$ into $\lceil m/2 \rceil$ sets, each of which will be assigned to a distinct queue. The ith set, $1 \leq i \leq \lceil m/2 \rceil$, comprises all edges of the form (a_i, b_j) and (a_{m+1-i}, b_j), for $1 \leq j \leq n$. Since none of the edges in the ith set nest, they can all be assigned to a single queue, whence $\lceil m/2 \rceil$ queues suffice.

Lower bound. Let σ be an order of the vertices in a $\mathrm{QN}(K_{m,n})$-queue layout of $K_{m,n}$. By symmetry, we may assume that the a_i's appear in the order $a_1, a_2, \cdots, a_m$ in σ and that the b_j's appear in the order $b_n, b_{n-1}, \cdots, b_1$ in σ. Because we may reverse σ and

still have a $\mathrm{QN}(K_{m,n})$-queue layout, we may assume that $b_{\lceil m/2 \rceil}$ appears after $a_{\lceil m/2 \rceil}$ in σ. Then the set of edges

$$\{(a_i, b_i) \mid 1 \leq i \leq \lceil m/2 \rceil\}$$

nest. By Proposition 2.1, $\mathrm{QN}(K_{m,n}) \geq \lceil m/2 \rceil$. □

This straightforward determination of $\mathrm{QN}(K_{m,n})$ contrasts with the current status of $\mathrm{SN}(K_{m,n})$ as reported in [20]. Even after much effort, the exact stacknumber of $K_{m,n}$, or even of $K_{n,n}$, has not been determined, though Muder, Weaver, and West [20] have obtained nontrivial bounds.

4.7. FFT and Beneš networks. We now consider two related families of graphs that have importance as computational networks. The *FFT network* represents the data dependencies of the Fast Fourier Transform algorithm. The *Beneš rearrangeable permutation network* is a switching network capable of realizing at its n outputs any permutation of its n inputs (Beneš [1]).

The n-input Beneš network $B(n)$, $n = 2^m$, is defined inductively as follows.

1. $B(2)$ is the complete bipartite graph $K_{2,2}$ on the two input vertices $I[1,1]$ and $I[1,2]$ and the two output vertices $O[1,1]$ and $O[1,2]$.

2. $B(n)$ is obtained from two copies of $B(n/2)$, together with n new input vertices $I[m,1], I[m,2], \cdots, I[m,n]$ and n new output vertices $O[m,1], O[m,2], \cdots, O[m,n]$. In the second copy of $B(n/2)$, each vertex $I[k,i]$ is relabeled $I[k, i+n/2]$, and each vertex $O[k,i]$ is relabeled $O[k, i+n/2]$; all vertices then have distinct labels. For $1 \leq i \leq n$, add edges to create a copy of $K_{2,2}$ on vertices $I[m,i]$ and $I[m, i+n/2]$ and vertices $I[m-1,i]$ and $I[m-1, i+n/2]$; also, add edges to create a copy of $K_{2,2}$ on vertices $O[m,i]$ and $O[m, i+n/2]$ and vertices $O[m-1,i]$ and $O[m-1, i+n/2]$.

As shown in Fig. 4.3, the Beneš network has a natural level structure with $2m$ levels. The n-input FFT network is the graph consisting of the first $m+1$ levels of $B(n)$.

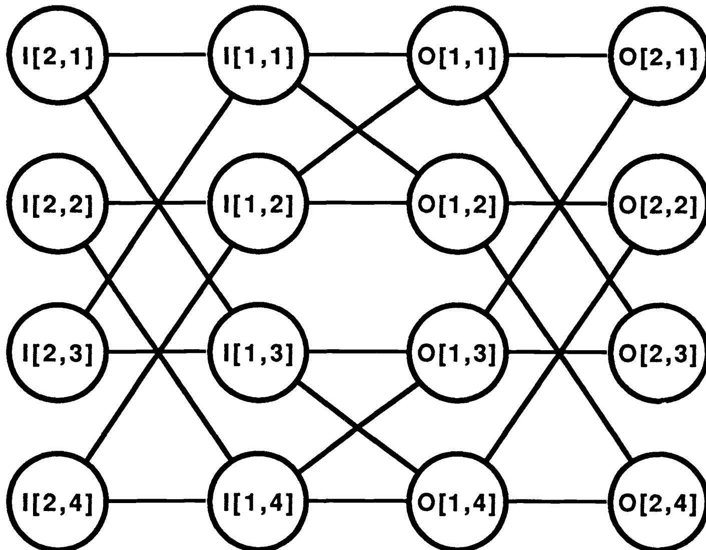


FIG. 4.3. *The Beneš network* $B(4)$.

As $B(n)$, $n > 2$, is not planar, its queuenumber is at least 2. The level structure (of either network) provides a straightforward 3-queue layout: order the vertices level by

level, going up each level; one queue for the "cross" edges, one queue for the "upward" edges, and one queue for the "downward" edges suffices. A more complicated 2-queue layout of $B(n)$ is due to Reibman [23].

PROPOSITION 4.9 (see [23]). *The Beneš network $B(n)$ admits a 2-queue layout with each queue of width n. The layout is optimal in queuenumber and within a factor of 2 of optimal in queuewidth.*

Proof. The layout of $B(n)$ follows its inductive definition. The inductive hypothesis is that $B(n)$ has a 2-queue layout which respects the leveling of $B(n)$; that is, all level i vertices appear before any level $i+1$ vertices, though no restriction is placed on the relative order of vertices within each level.

1. The vertex order for $B(2)$ is $I[1,1], I[1,2], O[1,1], O[1,2]$. The two edges incident to $I[1,1]$ are assigned to one queue and the two edges incident to $I[1,2]$ are assigned to the second queue. The layout satisfies the inductive hypothesis.

2. We assume that $B(n/2)$ has a 2-queue layout satisfying the inductive hypothesis. Let B_1 and B_2 be two copies of $B(n/2)$. Lay each out in the 2-queue order that is guaranteed by the induction. Merge the two layouts level by level so that the level-i vertices of B_2 always appear immediately to the right of the level-i vertices of B_1. In particular, $I[k, i+n/2]$ (respectively, $O[k, i+n/2]$) is always $n/2$ vertices to the right of $I[k,i]$ (respectively, of $O[k,i]$). Because the leveling of $B(n)$ is honored in the layout, each level-i edge of B_1 crosses every level-i edge of B_2, and vice versa, so no nesting results from the merging; hence, a 2-queue layout of the "sum" of B_1 and B_2 results. Add n new input vertices to the left and n new output vertices to the right of the entire layout. View the n new inputs as consisting of $n/2$ consecutive pairs of vertices. Add edges from the first pair to the first vertices of B_1 and B_2 to form a copy of $K_{2,2}$. In general, add edges from the ith pair to the ith vertices of B_1 and B_2. Assign the added edges incident to B_1 (which form a twist) to the first queue and the added edges incident to B_2 (which also form a twist) to the second queue. Similarly, connect the n new outputs to the last vertices of B_1 and B_2. The result is a 2-queue layout of $B(n)$. □

Because the FFT network is a subgraph of the Beneš network, it also has a 2-queue layout. This compares favorably with the stacknumber optimal 3-stack layouts of the Beneš and FFT networks in Games [8]. The natural leveling of these networks is a definite advantage in constructing queue layouts that are good, at least in the sense of queuenumber.

4.8. Hypercube. The *d-dimensional hypercube $Q(d)$* has vertex set $\{0,1\}^d$, the set of all bit strings of length d; its edges connect every pair of vertices that differ in exactly one bit position. View the vertex set of $Q(d)$ as the set of integers $\{0,1,\cdots,2^d-1\}$ by identifying a d-bit string with the corresponding integer in binary notation. The hypercube admits a very regular layout strategy.

PROPOSITION 4.10. *For $d \geq 2$, $Q(d)$ admits a $(d-1)$-queue layout with queuewidths*

$$2^{d-1}, 2^{d-2}, \cdots, 2^2, 2^1.$$

Proof. We lay out $Q(d)$ inductively. The order $\sigma = 0,1,2,3$ gives a 1-queue layout of $Q(2)$ with queuewidth 2. To obtain a layout for $Q(d)$, $d>2$, inductively lay out two adjacent copies of $Q(d-1)$, similarly ordered. By induction, each of the copies of $Q(d-1)$ uses $d-2$ queues with queuewidths

$$2^{d-2}, 2^{d-3}, \cdots, 2^2, 2^1;$$

hence, their disjoint sum does also. The 2^{d-1} edges connecting one copy of $Q(d-1)$ to the other form a 2^{d-1}-twist; hence they require only one additional queue of width 2^{d-1}. □

The queuenumber of the preceding layout is optimal to within a constant factor.

PROPOSITION 4.11. $\mathrm{QN}(Q(d)) = \Omega(d)$.

Proof. $Q(d)$ has $d2^{d-1}$ edges. By Corollary 3.7, therefore,

$$\mathrm{QN}(Q(d)) \geq \left\lceil \frac{d2^{d-1}}{2^{d+1}-3} \right\rceil = \Omega(d). \qquad \square$$

5. Queuenumber and graph structure. We now explore two structural properties of graphs that provide bounds on queuenumber. These properties are bandwidth and separator size.

5.1. Bandwidth. Let $\sigma = 1, 2, \cdots, n$ be any order of the vertices of G. The *bandwidth* of σ is the length of the longest edge; that is,

$$\mathrm{BW}(\sigma) = \max_{(i,j)\in E} |i-j|.$$

The *bandwidth* of G is the minimum bandwidth of any σ; that is,

$$\mathrm{BW}(G) = \min_{\sigma} \mathrm{BW}(\sigma).$$

Assume that $n \geq B+1$. The *maximal bandwidth-B graph on n vertices* $M(B,n)$ has vertex-set $\{1, 2, \cdots, n\}$; its edges form a copy of the complete graph K_{B+1} on each subset of vertices

$$\{i, i+1, \cdots, i+B\}, \quad 1 \leq i \leq n-B.$$

See Fig. 5.1.

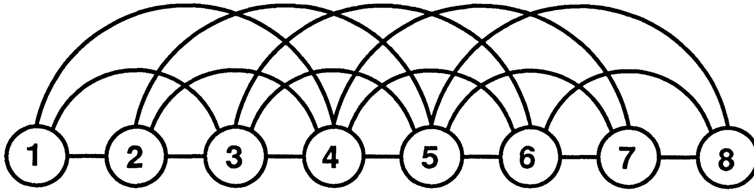


FIG. 5.1. *Maximal bandwidth-B graph* $M(3,8)$.

The following theorem establishes a relationship between bandwidth and queuenumber.

THEOREM 5.1. $\mathrm{QN}(M(B,n)) = \lceil B/2 \rceil$. *Hence, if* $\mathrm{BW}(G) = B$, *then* $\mathrm{QN}(G) \leq \lceil B/2 \rceil$.

Proof.

Upper bound. Choose the order $\sigma = 1, 2, \cdots, n$ for the vertices of $M(B,n)$. There are edges of every length from 1 to B. Assign all edges of lengths $2i-1$, $2i$, $1 \leq i \leq \lceil B/2 \rceil$, to queue q_i. No two edges having the same length or having lengths differing by 1 can nest. A $\lceil B/2 \rceil$-queue layout of $M(B,n)$ results.

Lower bound. $M(B,n)$ contains complete graphs on $B+1$ vertices. By Proposition 4.7,

$$\mathrm{QN}(M(B,n)) \geq \left\lfloor \frac{B+1}{2} \right\rfloor = \left\lceil \frac{B}{2} \right\rceil.$$

Since every bandwidth-B graph G is a subgraph of some maximal bandwidth-B graph $M(B,n)$, $\mathrm{QN}(G) \leq \lceil B/2 \rceil$. □

5.2. Separator size. Let $S(x)$ be a nondecreasing integer function. A graph $G = (V,E)$ has a $(\frac{1}{3},\frac{2}{3})$-*vertex-separator of size* $S(x)$ (or just *separator of size* $S(x)$) if either $|V| < 3$ or if there is a subset of cardinality at most $S(|V|)$ whose removal leaves connected components of cardinality less than or equal to $\frac{2}{3}|V|$, each having a separator of size $S(x)$.

Suppose G has maximum degree d and has a separator of size $S(x)$. Let R be the following function of S:

$$R(x) = (d+1)\sum_i S(x/2^i).$$

A *bucket tree for G* is a complete binary tree whose level-j *buckets* (vertices) have *bucket capacity*

$$C(j) = cdR(|V|/2^j),$$

where c is a constant. Bhatt et al. [3] demonstrate the following.

LEMMA 5.2 (see [3]). *If $G = (V,E)$ is a graph of maximum degree d that has a separator of size $S(x)$, then V can be mapped onto the bucket tree for G in such a way that*

1. *At most $C(j)$ vertices are mapped to each level-j vertex of the bucket tree;*
2. *If two vertices that are adjacent in G are mapped to two distinct buckets, then these two buckets are at most distance d apart in the bucket tree, and one of the buckets is an ancestor of the other.*

The following relates the separator size of a graph to its queuenumber and stacknumber.

THEOREM 5.3. *If $G = (V,E)$ is a graph of maximum degree d that has a separator of size $S(x)$, then G has queuenumber and stacknumber $O(d^2R(|V|))$.*

Proof.

Queuenumber. Construct a queue layout of G in two steps. First, use Lemma 5.2 to map V onto the bucket tree for G, and lay out the bucket tree in a breadth-first order. Second, use this 1-queue layout of the bucket tree to obtain a queue layout of G; replace each bucket B by the contents of B placed contiguously in any order. We analyze the number of queues needed in this layout of G. The two endpoints of any edge of G are mapped to a pair of buckets B_1 and B_2 such that B_1 is an ancestor of B_2, and B_1 and B_2 are at most distance d apart in the bucket tree; call an edge an i-*edge,* $0 \leq i \leq d$, if i is the distance between B_1 and B_2 in the bucket tree. If the endpoints of two i-edges are mapped to different pairs of buckets, then the two edges cannot nest. Fix i, $0 \leq i \leq d$. Since each bucket contains $O(dR(|V|))$ vertices, $O(dR(|V|))$ queues suffice for all i-edges. Since i has $d+1$ possible values, the queue layout for G has queuenumber $O(d^2R(|V|))$.

Stacknumber. A similar construction that begins by laying out the bucket tree in preorder suffices to show the bound on stacknumber. □

We remark that Theorem 4.5 of Chung, Leighton, and Rosenberg [5] gives an upper bound on the stacknumber of a graph as a function of its bifurcator, rather than separator, size.

6. A queuenumber/queuewidth tradeoff. In this section, we provide evidence of an apparent tradeoff between queuenumber and queuewidth for queue layouts of complete

binary trees. Then we relate the queuewidth of a graph G to the *diameter* of G, i.e., the greatest distance between any pair of vertices of G.

By Proposition 4.1, a depth-d complete binary tree has a 1-queue layout with queuewidth 2^d. This queuewidth is exponentially greater than the $O(d)$ stackwidth of a 1-stack layout of a complete binary tree [5]. We consider the question of whether a larger number of queues can be traded off for a smaller (cumulative) queuewidth. The following theorem suggests an apparent tradeoff.

THEOREM 6.1. *A depth-d complete binary tree T with $n = 2^d$ leaves has a k-queue layout QL with* $\mathrm{CQW}(QL) = O(kn^{1/k})$.

Proof. We give the proof for $k = 2$. To simplify the construction, we assume that $d = 2d'$ is even. Let $n' = 2^{d'} = \sqrt{n}$. Let $T^\star$ be the upper $d' + 1$ levels of T, and let $1, 2, \cdots, n'$ be the leaves of $T^\star$ in canonical order. Each leaf i is the root of a subtree T_i of depth d'. Order the vertices of $T^\star$ in breadth-first order from the root, so that vertices $1, \cdots, n'$ appear rightmost in the order. For each $i, 1 \leq i \leq n'$, place the vertices of T_i in breadth-first order immediately to the right of its root i. Assign the edges of $T^\star$ to one queue and the edges of $T_1, T_2, \cdots, T_{n'}$ to a second queue. Each queue has queuewidth $n' = \sqrt{n}$. The cumulative queuewidth of the 2-queue layout is $2n' = O(2n^{1/2})$.

For general k, $2 \leq k \leq d$, cut T every $\lceil d/k \rceil$ levels, and use one queue for the edges in each of the produced "meta-levels." Details are left to the reader. □

To show that the apparent tradeoff is a real one, we need lower bound techniques for queuewidth. The next theorem provides a lower bound on queuewidth as a function of diameter for arbitrary 1-queue graphs.

THEOREM 6.2. *Suppose $G = (V, E)$ is a connected 1-queue graph having diameter D. Let QL be a 1-queue layout of G. Then,*

$$\mathrm{QW}(QL) \geq \frac{|E|}{2D + 1}.$$

Proof. By Theorem 3.2, layout QL yields an arched leveled-planar embedding of G; denote the induced levels by $V_1, V_2, \cdots, V_m$. Since each edge of G connects vertices that are either in the same or adjacent levels, it is immediate that $D \geq m - 1$. Consider now the following $2m - 1 \leq 2D + 1$ cuts of layout QL:[1]

$$\mathrm{CUT}(t_1 - 1), \mathrm{CUT}(t_1), \cdots, \mathrm{CUT}(t_i - 1), \mathrm{CUT}(t_i), \cdots, \mathrm{CUT}(t_m - 1).$$

Since every edge of G either has some t_i as its right endpoint or passes over some t_i, the enumerated set of cuts collectively exhausts E. The proof is completed by appealing to two facts:

- The queuewidth of the layout QL is (clearly) as big as the biggest cut.
- The biggest cut contains no fewer edges than the average cut. □

For a depth-d complete binary tree T, there are $n = 2^d$ leaves, $|V| = 2n - 1$, $|E| = 2n - 2$ and $D = 2d = 2\log n$. We have this corollary.

COROLLARY 6.3. *Any 1-queue layout of a depth-d complete binary tree has queuewidth at least* $(2n - 2)/(4d + 1) = \Omega(n/\log n)$.

The breadth-first layout of T starting at the root has queuewidth n. The lower bound on queuewidth in the corollary is close to this upper bound. We do not know how to achieve this lower bound and doing so appears difficult. The higher width of queue

[1] $\mathrm{CUT}(i)$, the *cut* at vertex i, is the set of all edges whose left endpoint is less than or equal to i and whose right endpoint is greater than i.

layouts over stack layouts suggests that stack layouts of trees are preferable to queue layouts.

We do not yet have a lower bound on cumulative queuewidth for arbitrary k-queue layouts of T, along the lines of Theorem 6.2 for the case $k = 1$. Thus, we do not know whether there is a real tradeoff between queuenumber and queuewidth here, but we conjecture that there is. We further conjecture that the cumulative queuewidth announced in Theorem 6.1 is within a factor of $O(d)$ of optimal.

7. Future directions. In Table 7.1, we summarize our queuenumber results for specific graphs alongside the corresponding stacknumber results. Further comparison of the relative merits of queues and stacks is warranted. In particular, queues appear to be more appropriate than stacks for graphs with a leveled structure; can this insight be formalized? Chung, Leighton, and Rosenberg [5], and Heath [13] show that there are tradeoffs between stacknumber and stackwidth in the sense that, for certain graphs, devoting more stacks to a layout decreases the cumulative stackwidth. We expect analogous tradeoffs between queuenumber and queuewidth. We conjecture the following.

CONJECTURE 1. There are graphs that exhibit a tradeoff between queuenumber and queuewidth, and Theorem 6.1 exposes such a tradeoff.

TABLE 7.1
Queuenumbers of specific graphs.

Graph Class	Queuenumber	Stacknumber
Trees	1	1 [5]
X-trees	2	2 [5]
DeBruijn Graph	2	≤ 5 [21]
Complete Graph K_n	$\lfloor n/2 \rfloor$	$\lfloor n/2 \rfloor$ [5]
Complete Bipartite Graph $K_{m,n}$	$\min(\lceil m/2 \rceil, \lceil n/2 \rceil)$ (Exact)	$\leq \lceil (m+2n)/4 \rceil$ [20]
FFT Network	2	3 [8]
Beneš Network	2	3 [8]
Boolean n-cube	$\leq n-1$	$\leq n-1$ [5]
Ternary n-cube	$\leq 2n-2$ [16]	$\Omega(3^{\alpha n})$, $\alpha < 1/9$ [16]
Planar Graphs	Unknown (Conjecture bounded)	4 [27]

Often, good queue layouts seem easier to obtain than good stack layouts. Planar graphs may be an exception to this. However, in harmony with the fact that planar graphs can be laid out in a bounded number of stacks (Yannakakis [27]), we conjecture the following.

CONJECTURE 2. Planar graphs can be laid out in a bounded number of queues.

The notions of stack and queue layouts may be generalized in several directions. One approach is to define layouts that simultaneously utilize queues and stacks. We conjecture the following.

CONJECTURE 3. Each planar graph admits a 1-stack, 1-queue layout.

Another approach is to utilize deques or more general permutation mechanisms. In the realm of such generality, it becomes necessary to consider relative cost measures for the various mechanisms.

Note added in proof. The interested reader should be aware of the Ph.D. thesis of Sriram V. Pemmaraju (*Exploring the Powers of Stacks and Queues via Graph Layouts*, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1992). It contains further results in the theory of queue and stack layouts. The thesis develops a number of new tools for studying such layouts. It also initiates the study (promised in §1.2) of queue and stack layouts of *dags*. Copies are available from the first author.

Acknowledgments. Part of this research was conducted while the first author was at the University of North Carolina at Chapel Hill and at the Massachusetts Institute of Technology. A portion of the work of the second author was done during a visit to the Department of Applied Mathematics and Informatics of the University of the Saarlands.

We wish to thank Reuven Bar-Yehuda, Sandeep Bhatt, Fan Chung, Sergio Fogel, Tom Leighton, Bojana Obrenić, Sriram Pemmaraju, and Andrew Reibman for helpful conversations. We especially thank Bojana for a careful reading of the completed paper, for many useful suggestions, and for pointing out the construction in the proof of Theorem 5.3.

REFERENCES

[1] V. E. BENEŠ, *Optimal rearrangeable-multistage connecting networks*, Bell System Technical Journal, 43 (1964), pp. 1641–1656.

[2] F. BERNHART AND B. KAINEN, *The book thickness of a graph*, J. Combin. Theory B, 27 (1979), pp. 320–331.

[3] S. N. BHATT, F. R. K. CHUNG, J.-W. HONG, F. T. LEIGHTON, B. OBRENIĆ, A. L. ROSENBERG, AND E. J. SCHWABE, *Optimal emulations by butterfly-like networks*, Tech. Rpt. 90-108, Department of Computer Science, University of Massachusetts, Amherst, MA, 1990; J. Assoc. Comput. Mach., to appear.

[4] J. BUSS AND P. SHOR, *On the pagenumber of planar graphs*, in Proceedings of the 16th Annual ACM Symposium on Theory of Computing, Washington, DC, 1984, pp. 98–100.

[5] F. R. K. CHUNG, F. T. LEIGHTON, AND A. L. ROSENBERG, *Embedding graphs in books: A layout problem with applications to VLSI design*, SIAM J. Algebraic Discrete Meth., 8 (1987), pp. 33–58.

[6] P. ERDÖS AND E. SZEKERES, *A combinatorial problem in geometry*, Compositio Math., 2 (1935), pp. 463–470.

[7] S. EVEN AND A. ITAI, *Queues, stacks and graphs*, in Theory of Machines and Computations, Z. Kohavi and A. Paz, eds., Academic Press, New York, 1971, pp. 71–86.

[8] R. GAMES, *Optimal book embeddings of the FFT, Benes, and barrel shifter networks*, Algorithmica, 1 (1986), pp. 233–250.

[9] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability*, W. H. Freeman, New York, 1979.

[10] M. R. GAREY, D. S. JOHNSON, G. L. MILLER, AND C. H. PAPADIMITRIOU, *The complexity of coloring circular arcs and chords*, SIAM J. Algebraic Discrete Meth., 1 (1980), pp. 216–227.

[11] L. S. HEATH, *Embedding planar graphs in seven pages*, in Proceedings of the 25th Annual IEEE Symposium on Foundations of Computer Science, Singer Island, FL, 1984, pp. 74–83.

[12] ———, *Algorithms for Embedding Graphs in Books*, Ph.D. thesis, Department of Computer Science, University of North Carolina, Chapel Hill, NC, 1985.

[13] ———, *Embedding outerplanar graphs in small books*, SIAM J. Algebraic Discrete Meth., 8 (1987), pp. 198–218.

[14] L. S. HEATH AND S. ISTRAIL, *The pagenumber of genus g graphs is* $O(g)$, in Proceedings of the 19th Annual ACM Symposium on Theory of Computing, New York, NY, 1987, pp. 388–397.

[15] ———, *The pagenumber of genus g graphs is* $O(g)$, J. Assoc. Comput. Mach., 1992, to appear.

[16] L. S. HEATH, F. T. LEIGHTON, AND A. L. ROSENBERG, *Comparing queues and stacks as mechanisms for laying out graphs*, SIAM J. Discrete Math., 5 (1992), to appear.

[17] W.-L. Hsu, *Maximum weight clique algorithms for circular-arc graphs and circle graphs*, SIAM J. Comput., 14 (1985), pp. 224–231.

[18] D. B. Johnson, *A priority queue in which initialization and queue operations take $O(\log\log D)$ time*, Math. Systems Theory, 15 (1982), pp. 295–309.

[19] D. Lichtenstein, *Planar formulae and their uses*, SIAM J. Comput., 11 (1982), pp. 329–343.

[20] D. J. Muder, M. L. Weaver, and D. B. West, *Pagenumber of complete bipartite graphs*, J. Graph Theory, 12 (1988), pp. 469–489.

[21] B. Obrenić, *Embedding de Bruijn and shuffle-exchange graphs in five pages*, in Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures, Hilton Head, SC, 1991, pp. 137–146.

[22] C. H. Papadimitriou and M. Yannakakis, *Towards an architecture-independent analysis of parallel algorithms*, in Proceedings of the 20th Annual ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 510–513.

[23] A. Reibman, *DIOGENES layouts using queues*, typescript, Department of Computer Science, Duke University, Durham, NC, 1984.

[24] A. L. Rosenberg, *The DIOGENES approach to testable fault-tolerant arrays of processors*, IEEE Trans. Comput., C-32 (1983), pp. 902–910.

[25] M. M. Syslo and M. Iri, *Efficient outerplanarity testing*, Fund. Inform., 2 (1979), pp. 261–275.

[26] R. E. Tarjan, *Sorting using networks of queues and stacks*, J. Assoc. Comput. Mach., 19 (1972), pp. 341–346.

[27] M. Yannakakis, *Embedding planar graphs in four pages*, J. Comput. System Sci., 38 (1989), pp. 36–67.

SIAM J. COMPUT.
Vol. 21, No. 5, pp. 959–971, October 1992

© 1992 Society for Industrial and Applied Mathematics
009

OPTIMAL ON-LINE SIMULATIONS OF TREE MACHINES BY RANDOM ACCESS MACHINES*

MICHAEL C. LOUI† AND DAVID R. LUGINBUHL‡

Abstract. This paper shows that every tree machine of time complexity t can be simulated on-line by a *log-cost* random access machine (RAM) of time complexity $O((t \log t)/\log\log t)$. Using information-theoretic techniques, it is shown that this simulation is optimal. It is also shown that every tree machine can be simulated by a unit-cost RAM in real time.

Key words. Kolmogorov complexity, on-line simulation, random access machine, real-time simulation, time complexity, tree machine

AMS(MOS) subject classifications. 68Q05, 03D10, 03D15, 68Q30

1. Introduction. The random access machine (RAM) and the Turing machine are the standard models for sequential computation. Research into the use of time and space by these and other models gives us insight into their computational power. This research includes analyzing how two different models use time and space, and comparing time and space within a single model. Another avenue of investigation is determining how altering the definitions of time and space (for example, *log-cost* versus unit-cost) for a model affects its computational power. Slot and van Emde Boas [13], for example, showed how space equivalence of RAMs and Turing machines is affected by varying the definition of space complexity for RAMs.

A *tree machine* is a Turing machine whose worktapes are complete infinite rooted binary trees. It is a natural model of sequential computation on hierarchical structures such as heaps and balanced trees. Paul and Reischuk [10] used tree machines to investigate the relationships between time and space for random access machines and multidimensional Turing machines. They presented a simulation of a *log-cost* RAM of time complexity t by a tree machine of time complexity $O(t)$. They also showed that a tree machine of time complexity t can be simulated off-line by a unit-cost RAM of time complexity $O(t/\log\log t)$. Loui [8] showed that a multihead tree machine of time complexity t can be simulated on-line by a tree machine with only two worktape heads in time $O((t \log t)/\log\log t)$. Reischuk [11] and Loui [7] studied the relationship between tree machines and multidimensional Turing machines.

We begin by showing that every tree machine can be simulated by a unit-cost RAM in real time. We then present our main results, which concern on-line simulation of tree machines by *log-cost* RAMs. We show that every tree machine of time complexity t can be simulated on-line by a *log-cost* RAM of time complexity $O((t \log t)/\log\log t)$. Using the notion of incompressibility from Kolmogorov complexity [6], we show that this simulation is optimal. This appears to be the first application of Kolmogorov complexity to sequential RAMs. It is significant because few algorithms have been shown to be optimal.

All logarithms in this paper are taken to base 2.

---

*Received by the editors July 2, 1990; accepted for publication (in revised form) July 26, 1991. The views, opinions, and conclusions in this paper are those of the authors and should not be construed as an official position of the Department of Defense, U.S. Air Force, or other U.S. government agency.

†Department of Electrical and Computer Engineering, Coordinated Science Laboratory, and Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, Illinois 61801. Supported by the Office of Naval Research contract N00014-85-K-0570 and by National Science Foundation grant CCR-8922008.

‡Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio 45433.

2. Machine definitions. All machines that we consider have a two-way read-only input tape and a one-way write-only output tape. The principal differences in the machines are in their storage structures.

A *tree machine*, a generalization of a Turing machine, has a storage structure that consists of a finite collection of complete infinite rooted binary trees, called *tree worktapes*. Each cell of a worktape can store a 0 or 1. Each worktape has one head. A worktape head can shift to a cell's parent or to its left or right child. Initially, every worktape head is on the root of its worktape, and all cells contain 0.

Let W be a tree worktape. We fix a natural bijection between the positive integers and cells of W. We refer to the integer corresponding to a particular cell as that cell's *location*. Write cell(b) for the cell at location b. Define cell(1) as the root of W. Then cell($2b$) is the left child of cell(b) and cell($2b+1$) is the right child of cell(b).

Each step of a tree machine consists of reading the contents of the worktape cells and input cell currently scanned, writing back on the same worktape cells and (possibly) to the currently accessed output cell, and (possibly) shifting each worktape head and the input head. When the tree machine writes on the output tape, it also shifts the output head.

The *time complexity* $t(n)$ of a tree machine is defined in the natural way, namely, the maximum number of steps taken by the tree machine on inputs of size n.

The *depth complexity* of a tree machine is $d(n)$ if every worktape head remains within distance d of the root of its worktape on every input of size n. It is possible to limit the depth complexity of a tree machine with respect to its time complexity.

THEOREM 2.1 (see [8], [10]). *Every tree machine running in time $t(n)$ can be simulated on-line by a tree machine running in time $O(t(n))$ and depth $O(\log t(n))$.*

The random access machine (RAM) [2], [3], [5] consists of the following: a finite sequence of labeled instructions, a memory consisting of an infinite sequence of registers, indexed by nonnegative integer addresses (register $r(j)$ has address j), and a special register AC, called the accumulator, used for operating on data. Each register, including AC, holds a nonnegative integer; initially all registers contain 0. Let $\langle x \rangle$ denote the contents of register $r(x)$ and $\langle AC \rangle$ denote the contents of AC. Each cell on the input tape contains a symbol from a finite input/output alphabet. The following RAM instructions are allowed.

input. Read the current input symbol into AC and move the input head one cell to the right.

output. Write $\langle AC \rangle$ to the output tape and move output head one cell to the right.

jump θ. Unconditional transfer of control to instruction labeled θ.

jgtz θ. Transfer control to instruction labeled θ if $\langle AC \rangle > 0$.

load $=C$. Load integer C into AC.

load j. Load $\langle j \rangle$ into AC.

load $*j$. (Load indirect) Load $\langle\langle j \rangle\rangle$ into AC.

store j. Store $\langle AC \rangle$ into $r(j)$.

store $*j$. (Store indirect) Store $\langle AC \rangle$ into register $r(\langle j \rangle)$.

add j. Add $\langle j \rangle$ to $\langle AC \rangle$ and place result in AC.

sub j. If $\langle j \rangle > \langle AC \rangle$, then load 0 into AC; otherwise, subtract $\langle j \rangle$ from $\langle AC \rangle$ and place result in AC.

The *length* of a nonnegative integer i is the minimum positive integer w such that $i \le 2^w - 1$ (approximately the logarithm of i). The binary representation of i has w bits.

We consider two time complexity measures for RAMs, based on the cost of each RAM instruction. For the *unit-cost* RAM, we charge each instruction one unit of time. For the *log-cost* RAM, we charge each instruction according to the *logarithmic cost criterion* [3]; the time for each instruction is the sum of the lengths of the integers (addresses and register contents) involved in its execution. The *time complexity* $t(n)$ of a RAM is the maximum total time used in computations on inputs of length n. It is possible, of course, to define time complexity in other ways; e.g., we could charge some other function $f(j)$ for access to register j [1].

In our simulations, we group the registers into a finite number of memories, each memory containing an infinite number of registers. This does not increase the cost in time by more than a constant factor, since we could simply interleave these memories into one memory [5].

Two RAM operations used in this paper are the *pack* and *unpack* operations. Let $r_1, r_2, \cdots, r_b$ be contiguous registers in RAM R's memory containing, respectively, x_1, $x_2, \cdots, x_b$, where each x_i is a single bit. R *packs* $r_1, r_2, \cdots, r_b$ by computing the single b-bit value $2^{b-1}x_1 + 2^{b-2}x_2 + \cdots + x_b$ and placing this value into the accumulator. The *unpack* operation is the inverse of the pack operation; R takes a single value in the accumulator and stores its bits into contiguous registers. Each operation has as parameters the beginning and ending addresses of the registers involved in the operation.

We use a technique of Katajainen, van Leeuwen, and Penttonen [5] to pack and unpack registers in order to find the bit representation of a number and vice-versa. This divide-and-conquer strategy involves precomputed shift tables.

LEMMA 2.2 (see [5]). *If the proper tables are available, then it is possible to compute the u-bit representation of an integer $n < 2^u$, and the numeric value of a u-bit string, both in $O(u \log u)$ time on a log-cost RAM.*

LEMMA 2.3 (see [5]). *The tables necessary for Lemma 2.2 can be built in $O(u2^u)$ time on a log-cost RAM.*

A machine M of time complexity t is simulated by a machine M' *on-line* in time $f(t)$ if for every time step t_i, where M reads/writes a symbol, there is a corresponding time step t'_i where M' reads/writes the same symbol, and $t'_i \leq f(t_i)$.

3. Simulation by unit-cost RAMs.

THEOREM 3.1. *Every tree machine can be simulated by a unit-cost RAM in real-time.*

Proof sketch. We design a unit-cost RAM R that simulates tree machine T with worktape W. R has a *contents memory*, a *parent memory*, and several working registers. Let *contents*(x) (respectively, *parent*(x)) be the register with address x in the contents (respectively, parent) memory. *Contents*(x) at address x contains the contents of cell(x) at location x in the worktape of T. If cell(x) is visited by T, then *parent*(x) contains the worktape location of the parent of cell(x). The working registers are used as temporary storage and to keep track of which cell is currently accessed by T.

R simulates one step of T with a constant number of accesses to the two memories and the working registers. For example, if the head moves from cell(x) to a child of cell(x), then R computes location $2x$ for the left child or $2x + 1$ for the right child with one or two additions and stores x in *parent*$(2x)$ or *parent*$(2x + 1)$. Thus to simulate t steps of T takes $O(t)$ time on R. □

4. Simulation by *log-cost* RAMs.

4.1. Upper bound. By Theorem 2.1, without loss of generality, we may assume that every head of T remains within distance $O(\log\ t)$ of the root. Thus every address used by

R has length $O(\log t)$, and under the *log-cost* measure, R runs in time $O(t \log t)$; however, we describe below a more efficient simulation by *log-cost* RAMs.

For simplicity, we consider tree machines with only one worktape, but our results generalize to multiple worktapes. Let T be a tree machine of time complexity t with one worktape W. We show that there is a RAM R that simulates T on-line in time $O((t \log t)/\log \log t)$.

Since this is an on-line simulation, we do not know n or $t(n)$ ahead of time. To solve this problem, we use a technique of Galil [4], adopted by Loui [7], [8] and Katajainen, van Leeuwen, and Penttonen [5]. Let t' be the elapsed time of T (as recorded by R), and let t_e be R's current estimate of the total running time of T. R begins the simulation with $t_e = 2$. When t' exceeds t_e, R doubles t_e and restarts the entire simulation. R continues this process of doubling t_e whenever t' exceeds t_e until the simulation is finished. R records the input in a separate memory as described below so that for each value of $t_e > 2$, it is unnecessary to move the input head until $t' > t_e/2$. We show that for each value of t_e, the time of the simulation is $O(t_e(\log t_e)/\log \log t_e)$. It is easy to show that the sum of the simulation times for all values of t_e is $O(t(\log t)/\log \log t)$, since the total running time for T is t.

We first provide a brief description of the simulation. We choose parameters h and u such that $u = 2^{2h+2} - 1$. We specify the values of h and u later. As noted earlier, R has several memories. R maintains in the *main memory* the entire contents of W. The main memory represents W as overlapping subtrees, called *blocks*. R represents the contents of each block W_x in one register r_x of the main memory. When the worktape head is in a particular block W_x, R represents W_x in the *cache memory*. Step-by-step simulation is carried out in the cache, which represents the block W_x in breadth-first order, one cell of W_x per register of the cache.

Because blocks overlap, when the worktape head exits W_x, it is positioned in the middle of some other block W_y. At this time R packs the contents of the cache back into r_x in the main memory and unpacks the contents of r_y into the cache.

The details of the simulation follow.

Let $W[x, s]$ be the complete subtree of W of height s rooted at cell(x). A *block* is any subtree $W_x = W[x, 2h+1]$ such that the depth of cell(x) is a multiple of $h+1$. Since a block has height $2h + 1$, it contains $2^{2h+2} - 1 = u$ cells. Let the *relative location* of a cell within a block be defined in a manner similar to the location of a cell, where the relative location of the root of the block is 1, the relative locations of its children are 2 and 3, and so on.

Call a block W_p the *parent block* of W_x if cell(p) is the ancestor of cell(x) at distance $h+1$ from cell(x). If W_x is the parent block of W_c, then W_c is a *child block* of W_x. Each block has 2^{h+1} child blocks. The topmost block of W, which contains the root of W, is called the *root block*.

Define the *top half* of a block W_x as $W[x, h]$, and define the *bottom half* of W_x as the remaining cells of the block. Note that the top half of the block W_x is part of the bottom half of W_p, its parent block, so that the blocks overlap. Call the portion of W_x shared by W_p (i.e., the subtree $W[x, h]$) the *common subtree* of W_x and W_p.

R precomputes in separate memories two tables, *half* and *translate*. We explain later how R uses these tables. Here we describe their contents and how they are computed. Let *half*(z) (respectively, *translate*(z)) be the register in *half* (respectively, *translate*) at address z.

Half(z) contains $\lfloor z/2 \rfloor$. To compute *half*, for $z = 1, \cdots, u/2$, R stores z in *half*$(2z)$ and *half*$(2z + 1)$.

For $z = 2^{2h+1}, \cdots, u$, *translate*(z) contains $(z \bmod 2^{h+1}) + 2^{h+1}$. R never refers to any register in *translate* with address less than 2^{2h+1}. *Translate* is computed as follows:

$i := 2^{h+1}$
for $z = 2^{2h+1}$ **to** u **do**
 translate$(z) := i$
 $i := i + 1$
 if $i = 2^{2h+2}$ **then** $i := 2^{h+1}$.

We now show how R simulates the tree machine using the cache. Assume the head of T is currently scanning a cell in block W_x. Let *cache*(z) be the register in the cache with address z, and let cell(x, z) be the cell in W_x with relative location z. For each $z = 1, \cdots, u$, register *cache*(z) contains the bit in cell(x, z); for example, *cache*(1) contains the contents of cell$(x, 1)$ = cell(x), the root of W_x. Thus R uses u registers of the cache, each register containing one bit.

While the head of T remains in W_x, R keeps track of the head's location with the *cache address register* in the *working memory*, a memory maintained by R for storing information necessary for miscellaneous tasks. If the cache address register contains z, then cell(x, z) is currently being accessed in T.

To simulate a tree machine operation at cell(x, z), R loads the contents (one bit) of *cache*(z) into AC. Once the contents are in AC, R simulates one step of T by storing either 0 or 1 in *cache*(z).

If the head of T moves to a child of cell(x, z), then the new address for the cache address register, as well as the relative location of the new block cell being read, is either $2z$ or $2z + 1$. With one or two additions, R computes this new address and places it in the cache address register. When the head of T moves to the parent of cell(x, z), the address of the corresponding cache register is $\lfloor z/2 \rfloor$. Because R has no division operation, it accesses the proper register of table *half* to retrieve the new address in cache.

To describe what happens when the worktape head moves out of the current block, we first show how the blocks are stored in main memory. Main memory is divided into *pages* consisting of $2^{h+1} + 3$ registers each. A page corresponds to a visited block of W. Let *page*(x) be the page representing W_x. Define the address of a page to be the address of the first register in the page. The first register in *page*(x) is the *contents register*. For the page representing the root block, the contents register contains the entire contents of that block. For every other block W_y, the contents register contains the contents of the bottom half of W_y. The contents of cells in a block are kept in breadth-first order; i.e., reading the binary string in the contents register from left to right is equivalent to reading the bottom half of the block it represents in breadth-first order. Initially, all cells of a block contain 0, so all contents registers initially contain 0.

Following the contents register is the *rank register*, containing a number ℓ between 1 and 2^{h+1}, indicating that W_x is the ℓth child of its parent block. The next register is the *parent register*, containing the address of the page representing the parent block of W_x. The next 2^{h+1} registers are the *child registers* of W_x. The mth child register of *page*(x) contains the address of the page representing the mth child block of W_x or 0 if that child block has not been visited (see Fig. 1).

The first page in main memory corresponds to the root block. Blocks are then stored in the order in which they are visited. The *page address register*, a register in working memory, contains the address of the page in main memory corresponding to the currently accessed block.

Let W_x be the currently accessed block and let W_p be the parent block of W_x. When

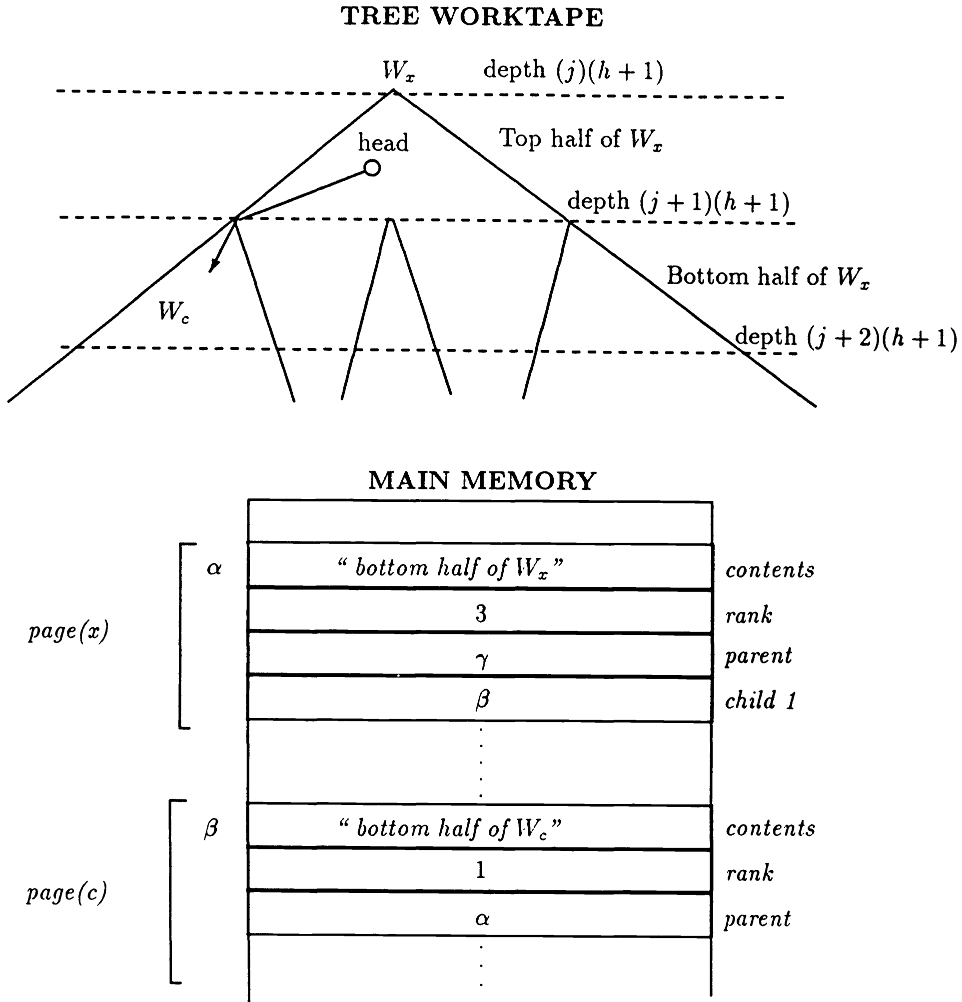


FIG. 1. *Worktape W (head moves from W_x to W_c).*

the tree worktape head moves out of W_x so that it is positioned in the middle of a child block W_c, R makes the proper changes to main memory and loads the cache from the contents register of *page*(c).

In main memory, R updates the contents registers of *page*(x) and *page*(p). To update *page*(x), R packs the contents of the registers of the cache that correspond to the bottom half of W_x into a single register in working memory (call it the *transfer register*, denoted by *tr*). R then copies *tr* into the contents register of *page*(x) via AC (see Fig. 2).

Updating *page*(p) consists of changing the bits of its contents register corresponding to the common subtree of W_x and W_p. R first saves the contents of the cache that encode the common subtree of W_x and W_c in a portion of working memory, since this information is needed in the cache as the top half of W_c. R also saves the contents of the cache that encode the common subtree of W_x and W_p. R then loads the contents register of *page*(p) into *tr* and unpacks the contents into the cache. The bits in working memory corresponding to the common subtree of W_x and W_p are then written into their

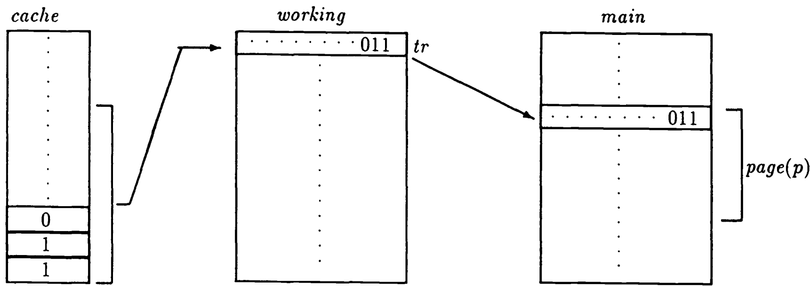


FIG. 2. *Updating* page(p) *in main memory.*

proper locations in the portion of the cache representing the bottom half of W_p. R then packs the contents of the cache into *tr* and copies *tr* into the contents register of *page*(p).

R then determines whether W_c has been visited before by checking the contents of the child register of *page*(x) corresponding to W_c. If the child register contains a valid (i.e., nonzero) address, then R uses that address to access *page*(c). R then loads the contents register of *page*(c) into the cache. This action is similar to the manipulation of *page*(p) discussed above. R loads the contents of the common subtree of W_x and W_c saved in working memory into the registers of the cache representing the top half of the block.

If the child register of *page*(x) contains 0, then R allocates a new page to maintain the information on W_c.

R modifies the page address register to reflect the fact that the worktape head is now scanning block W_c. The address currently in this register is that of *page*(x). R writes the address of *page*(c) in main memory into the page address register. R determines from the cache address register the quantity ℓ such that W_c is the ℓth child of W_x. Then by accessing the ℓth child register of *page*(x) in the main memory, R can determine the address of *page*(c).

To modify the cache address register to reflect the relative location of the head within block W_c, R first translates the relative location of the leaf cell(x, z) in W_x to its relative location in W_c. Since leaf cell(x, z) in W_x is the same as cell($c, (z \bmod 2^{h+1}) + 2^{h+1}$) in W_c, R uses the table *translate* described above. Using one or two additions, R then calculates the relative location in W_c of this cell's left or right child, depending on which branch the worktape head used to exit W_x. R then writes this new relative location into the cache address register.

A similar sequence of operations occurs if the worktape head moves out of a block (and further) into its parent block instead of into a child block. Then R uses the parent register to determine the address of the page representing the parent block, and R uses the rank register to determine the relative location of the worktape head within the parent block.

As described earlier, R maintains an estimate t_e of the total running time of T. R doubles t_e whenever the elapsed time exceeds t_e and restarts the simulation with this new value. The portion of the input string read by T up to time $t_e/2$ is maintained in R's *input memory* in registers of length h. Input symbols read from time 1 to time h are contained in the first register of input memory; those read from time $h+1$ to time $2h$ are contained in the second register, etc. Each register is unpacked into the *input cache*

at its appropriate time, and the input symbols are read by R. After $t_e/2$ steps of the tree machine have been simulated, input is read from the input tape. This new input is stored in the same manner as previous input.

When it is necessary to restart the simulation with a new value of t, R reorganizes the input memory using packs and unpacks so that register lengths reflect the updated value of h.

To simulate tree machines with more than one worktape, R maintains a main memory, a cache, and a working memory for each worktape.

By evaluating the cost of the simulation on a *log-cost* RAM, we derive the following result.

THEOREM 4.1. *Every tree machine running in time $t(n)$ can be simulated on-line by a log-cost RAM running in time $O((t(n)\log t(n))/\log\log t(n))$.*

Proof. Because the blocks have height $2h+1$ and overlap by height $h+1$, each time the worktape head moves out of a block it is exactly in the middle of another block; i.e., it will take at least $h' = h+1$ steps before it exits this new block. Since the tree machine computation has at most t steps, the work of updating main memory from cache (packing), loading a new block into cache (unpacking), and directly simulating h' steps is performed at most t/h' times.

Updating main memory and loading a new block into cache involve the pack and unpack operations and a constant number of accesses to main memory. By Lemma 2.2, the time for each pack and unpack operation is $O(u\log u)$. Since registers in main memory have addresses no larger than $(t/h')(2^{h+1}+3)$, each access to main memory takes time $O(\log t + h)$.

By Lemma 2.3, the time to create the tables necessary for the pack and unpack operations is $O(u2^u)$. The time to compute tables *half* and *translate* is $O(u\log u)$.

Simulating one step of the tree machine consists of a constant number of accesses to cache, taking time $O(\log u)$. Thus simulating h' steps takes time $O(h'\log u)$.

Simulating h input operations (those up to step $t/2$) takes time $O(h\log h)$. Recording h input operations (those past step $t/2$) also takes time $O(h\log h)$. Packing and unpacking take time $O(h\log h)$. Thus the time to simulate $t/2$ input operations and record $t/2$ additional input operations is $(t/h)O(h\log h)$. Reconfiguring the input memory for a new value of t also takes time $(t/h)O(h\log h)$. Building the necessary tables for input simulation and recording takes time $O(h2^h)$.

The total time required for R, then, is

$$(t/h')(O(\log t + h) + O(u\log u) + O(h'\log u)) + O(u2^u) + O(t\log h).$$

Since $h = O(\log u)$, the total time is

$$O(((t\log t)/\log u) + tu + t\log u + u2^u).$$

Choose h so that $u = (\log t)/\log\log t$. Then the total time for the simulation is $O((t\log t)/\log\log t)$. □

4.2. Lower bound. We now show that the time bound of Theorem 4.1 is optimal within a constant factor. We begin with an overview of Kolmogorov complexity, which we use to prove the lower bound.

Let σ and τ be strings in $\{0,1\}^*$, and let U be a universal Turing machine. Define the *Kolmogorov complexity* of σ given τ with respect to U, denoted $K(\sigma|\tau)$, as follows: let # be a symbol not in $\{0,1\}^*$; then $K(\sigma|\tau)$ is the length of β, where β is the shortest

binary string such that $U(\beta\#\tau) = \sigma$. Informally, $K(\sigma|\tau)$ is the length of the shortest binary description of σ, given τ. If τ is the empty string, then we write $K(\sigma)$ for $K(\sigma|\tau)$.

We say a string σ is *incompressible* if $K(\sigma) \geq |\sigma|$. Note that for all n there are 2^n binary strings of length n, but there are only $2^n - 1$ strings of length less than n. Thus for all n, there is at least one incompressible string of length n.

A useful concept in Kolmogorov complexity is the *self-delimiting string*. For natural number n, let *bin*(n) be the binary representation of n without leading zeros. For binary string w, let $\overline{w}$ be the string resulting from placing a 0 between each pair of adjacent bits in w and adding a 1 to the end. Thus $\overline{110} = 101001$. We call the string $\overline{bin(|w|)}w$ the *self-delimiting version* of w. The self-delimiting version of w has length $|w| + 2\lceil\log(|w|+1)\rceil$. When we concatenate several binary string segments of differing lengths, we can use self-delimiting versions of the strings so that we can determine where one string ends and the next string begins with little additional cost in the length of the concatenated string. Note that in such a concatenation it is not necessary to use a self-delimiting version of the last string segment.

Kolmogorov complexity has recently gained popularity as a method for proving lower bounds. Li and Vitanyi [6] provide a thorough summary of lower bound (and other complexity-related) results obtained using Kolmogorov complexity.

THEOREM 4.2. *There is a tree machine T running in time n such that for any log-cost RAM R, R requires time $t(n) = \Omega((n \log n)/\log\log n)$ to simulate T on-line.*

Proof. For simplicity, we omit floors and ceilings in this proof.

Tree machine T has one tree worktape and operates in real time. T's input alphabet is a set of *commands* of the form $\langle e, \psi\rangle$, where $e \in \{0, 1, ?\}$ and ψ indicates whether the worktape head moves to a child or parent of the current cell or remains at the current cell. Suppose T is in a configuration in which the cell x at which the worktape head is located contains e'. On input $\langle e, \psi\rangle$, machine T writes e' onto its output tape, and the worktape head writes e onto cell x if $e \in \{0, 1\}$, but it writes e' (the current contents of x) onto x if $e = ?$. At the end of the step the worktape head moves according to ψ. For every n that is a sufficiently large power of 2, we construct a series of n tree commands for which R requires time $\Omega((n \log n)/\log\log n)$. As in [7], the string of tree commands is divided into a *filling part* of length $n/2$ and a *query part* of length $n/2$.

Let W be the worktape of T, and let x_0 be the root of W. Let $d = \log(n/8)$. Denote the complete subtree of W of height d whose root is x_0 by W_d. Let $N = n/8$. We consider the complexity of the simulation in terms of N.

We fill W_d with an incompressible string τ of length $2N - 1$ such that τ can be retrieved by a depth-first traversal of W_d. This takes time $4N - 4$ on T. We move the worktape head four more times (without writing) so that the total length of the filling part is $n/2$.

The query part consists of a series of *questions*. A *question* is a string of $2d = 2\log N$ tree commands that causes the worktape head to move from the root x_0 of the tree worktape to a cell at depth d and back to x_0 without changing the contents of the worktape. As the head visits each cell during a question, T outputs the contents of that cell. T processes $2N/\log N$ questions $Q_1, Q_2, \cdots$ during the query part. Thus the query part takes time $4N = n/2$. We show that after each question Q_j, there is a question Q_{j+1} such that R takes time $\Omega((\log^2 N)/\log\log N)$ to process Q_{j+1}, and Theorem 4.2 follows.

Assume that R has just processed question Q_j. Let $P(N)$ be the maximum time necessary to process any possible next question. We show that some next question takes time $\Omega((\log^2 N)/\log P)$. Consequently, by definition, $P = \Omega((\log^2 N)/\log P)$. To determine a lower bound on P, we consider two cases:

(1) $P \leq \log^2 N$; hence, $\log P \leq 2 \log\log N$. Thus we have the following:

$$P \geq c(\log^2 N)/\log P, \text{ for some constant } c \text{ (since } P = \Omega((\log^2 N)/\log P))$$
$$\geq c(\log^2 N)/(2\log\log N);$$

(2) $P \geq \log^2 N$.
In either case, $P = \Omega((\log^2 N)/\log\log N)$.

We first determine $\hat{t}$, the sum over all possible next questions q, of the time required for R to process q.

Divide worktape W into $S = (\log N)/(2\log P)$ *sections*, each of height $2\log P$. For $s = 0, 1, \cdots, S-1$, there are P^{2s+2} exit points (*bottom cells*) in section s. We refer to any initial segment of a question as a *partial question* and the portion of the question that is processed while the worktape head is in one section as a *subquestion* (see Fig. 3). To compute $\hat{t}$, we compute for $s = 0, 1, \cdots, S-1$ the total time $\hat{t}_s$ required for R to process all possible subquestions in section s. Since the depth of W_d is $\log N$, there are N possible next questions. Each of the P^{2s+2} bottom cells of section s is visited during N/P^{2s+2} of these questions.

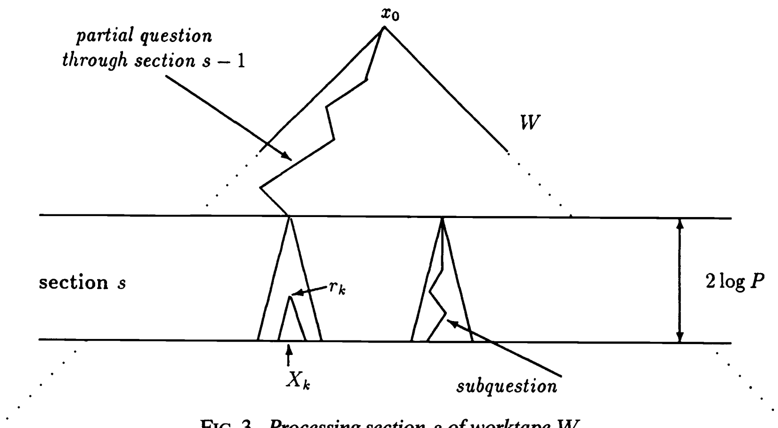


FIG. 3. *Processing section s of worktape W.*

Let σ_s be the string defined by the contents of the bottom cells of section s, from left to right; clearly, $|\sigma_s| = P^{2s+2}$.

LEMMA 4.3. *The string σ_s is incompressible up to a term of $O(s\log P)$; i.e., $K(\sigma_s) \geq |\sigma_s| - O(s\log P)$.*

Proof. The incompressible string τ, which gives the contents of W, can be specified by a string composed of the following segments:

1. a self-delimiting string encoding this discussion ($O(1)$ bits),
2. a self-delimiting version of a binary string of length $K(\sigma_s)$ that specifies σ_s ($K(\sigma_s) + O(s\log P)$ bits),
3. self-delimiting versions of the values of s and P ($O(\log s) + O(\log P)$ bits),
4. a string specifying the bits in τ but not in σ_s ($2N - 1 - P^{2s+2}$ bits).

Thus $K(\tau) \leq K(\sigma_s) + (2N - 1 - P^{2s+2}) + O(s\log P)$. But $K(\tau) \geq 2N - 1$; therefore, $K(\sigma_s) \geq P^{2s+2} - O(s\log P)$.

LEMMA 4.4. *If $\ell \geq 1$, then $\sum_{i=1}^{\ell} \log i \geq (1/2)\ell\log\ell$.*

Proof. For all i such that $1 \leq i \leq \ell$, clearly $(i-1)(\ell - i) \geq 0$; hence $i(\ell - i + 1) \geq \ell$. Consequently,

$$\begin{aligned}\sum_{i=1}^{\ell}\log i &= (1/2)\sum_{i=1}^{\ell}(\log i+\log(\ell-i+1))\\ &= (1/2)\sum_{i=1}^{\ell}\log(i(\ell-i+1))\\ &\geq (1/2)\sum_{i=1}^{\ell}\log \ell\\ &= (1/2)\ell\log\ell.\end{aligned}$$

LEMMA 4.5. *For $s = 1, 2, \cdots, S-1$, the number of registers accessed during the processing of all partial questions through section $s-1$ is at most $4P^{2s+1}/\log P$.*

Proof. Let $C = 4P/\log P$. By Lemma 4.4, for P sufficiently large, $\sum_{i=1}^{C}\log i \geq P$. The processing of each partial question through section $s-1$ could involve no more than C distinct registers; otherwise, because of the total cost of addresses of registers, R would exceed time P for some next question. There are P^{2s} different partial questions possible through section $s-1$, so there are no more than $4P^{2s+1}/\log P$ registers accessed for all possible partial questions.

Let us consider a particular section s. Let $r_1, r_2, \cdots, r_m$ be the registers, in order of increasing address, that R accesses to produce the same output that T produces when its worktape head is in section s, excluding those registers accessed to process partial questions through section $s-1$. The address of r_i is at least i. To compute a lower bound on $\hat{t}_s$, we assess for each i the contribution to $\hat{t}_s$ of accessing r_i.

To determine the contribution of r_i to $\hat{t}_s$, we calculate the minimum number of possible questions for which R accesses r_i. For every bottom cell v, let q_v be the partial question that causes T to visit cell v of the tree worktape. For $1 \leq i \leq m$, let X_i be the set of bottom cells x of section s such that $x \in X_i$ if R accesses r_i to process q_x (see Fig. 3). Thus if T visits a cell in X_i when processing a question in section s, R accesses register r_i when processing the same question. We say that r_i *operates* on the bottom cells in X_i. Since T visits one cell of X_i while processing one of N/P^{2s+2} possible questions, R accesses r_i during the processing of at least $|X_i|(N/P^{2s+2})$ possible questions.

For $1 \leq i \leq m$, the total access time for register r_i in section s is at least the product of $\log i$ (since the address of r_i is at least i), $|X_i|$ (the number of bottom cells that r_i operates on), and N/P^{2s+2} (the number of questions during which one of these bottom cells is visited). Summing the time incurred by access to each register yields

$$\hat{t}_s \geq \sum_{i=1}^{m}(\log i)|X_i|(N/P^{2s+2}). \tag{4.1}$$

Using Lemma 4.7 below, we can determine a lower bound for $\hat{t}_s$, but we first introduce the following technical lemma.

LEMMA 4.6 (see [9]). *Let J and M be integers such that $M \geq J$. A sorted J-member subset of $\{0, \cdots, M\}$ can be represented with no more than $2J\log(M/J) + 4J + 2$ bits.*

Let $h = (1/7)P^{2s+1}$.

LEMMA 4.7. $\sum_{i=h}^{m}|X_i| \geq (1/23)P^{2s+2}$.

Proof. Assume that the conclusion is false. Then $r_1, \cdots, r_{h-1}$ operate on at least $(22/23)P^{2s+2}$ bottom cells in section s. We can specify the string σ_s as follows: we obtain the bits of $X_h, \cdots, X_m$ explicitly. We obtain the other bits of σ_s by simulating R on each partial question to a bottom cell of section s not in $\bigcup_{k=h}^{m} X_k$. On each such partial question, R uses only registers $r_1, \cdots, r_{h-1}$ and registers accessed in sections $1, \cdots, s-1$. Thus σ_s can be specified with a string composed of the following segments:

1. a self-delimiting string encoding the program of R and this discussion ($O(1)$ bits),
2. self-delimiting versions of the addresses and initial contents of registers accessed in sections $1, \cdots, s-1$ (at most $8P^{2s+2}/\log P + O(s \log P)$ bits – by Lemma 4.5, at most $4P^{2s+1}/\log P$ registers are required, and for each register, the contents and the address could each require P bits),
3. self-delimiting versions of the addresses and initial contents of $r_1, \cdots, r_{h-1}$ (at most $(2/7)P^{2s+2} + O(s \log P)$ bits),
4. a string specifying positions of cells in X_k for $k \geq h$ (we use Lemma 4.6 with $J = (1/23)P^{2s+2}$ and $M = P^{2s+2}$; this requires at most $(14/23)P^{2s+2}$ bits, and the encoding used to achieve Lemma 4.6 is such that the beginning and end of this string can easily be determined),
5. a string specifying the contents of cells in X_k for $k \geq h$ (at most $(1/23)P^{2s+2}$ bits).

This means that the number of bits needed to specify σ_s is at most $(151/161)P^{2s+2} + O(P^{2s+2}/\log P) < P^{2s+2} - O(s \log P)$ for sufficiently large P. Thus we have a contradiction of Lemma 4.3.

Thus we have

$$\begin{aligned}
\hat{t}_s &\geq \sum_{i=1}^{m}((\log i)|X_i|(N/P^{2s+2})) && \text{(Inequality (4.1))}\\
&\geq \sum_{i=h}^{m}((\log i)|X_i|(N/P^{2s+2}))\\
&\geq (N/P^{2s+2})(\log h)\sum_{i=h}^{m}|X_i|\\
&\geq (N/P^{2s+2})(\log h)(1/23)P^{2s+2} && \text{(Lemma 4.7)}\\
&\geq (1/23)N((2s+1)\log P - \log 7) && \text{(definition of } h\text{)}\\
&\geq (1/23)Ns\log P.
\end{aligned}$$

Now sum $\hat{t}_s$ over all s to compute a lower bound for $\hat{t}$, the total time required for R to process all possible next questions:

$$\begin{aligned}
\hat{t} &= \sum_{s=0}^{S-1}\hat{t}_s\\
&\geq \sum_{s=0}^{S-1}((1/23)Ns\log P)\\
&\geq (1/23)N(\log P)((\log^2 N)/(4\log^2 P) - O((\log N)/\log P))\\
&\geq (1/92)((N\log^2 N)/\log P - O(\log N)).
\end{aligned}$$

Since there are N questions, we divide $\hat{t}$ by N to derive the average time needed by R to process the next question, $\Omega((\log^2 N)/\log P)$. Some next question must require time greater than or equal to this average time. Since P is the maximum time for some next question, $P \geq \Omega((\log^2 N)/\log P)$; hence, $P = \Omega((\log^2 N)/\log\log N)$.

Thus for each question Q_j, we can choose a next question Q_{j+1} that takes time $\Omega((\log^2 N)/\log\log N)$. Since the query part has $N/(2\log N)$ questions, our choice of questions means the query part takes time $t = (N/(2\log N))\Omega((\log^2 N)/\log\log N) = \Omega((N\log N)/\log\log N)$. The entire simulation takes at least time t. Since $N = n/8$, the lower bound holds for n in place of N. □

Because the lower bound proof considers only the time involved in accessing registers, the lower bound holds for RAMs with more powerful instructions, such as Boolean operations and multiplication.

5. Conclusions. Because the *log-cost* RAM is considered a "standard" among models of computation, it is important to determine its relationships to other models. Here we have shown an optimal on-line relationship between *log-cost* RAMs and tree machines. We hope that this work will lead to further study of relationships between other models of computation.

Some further areas of research include:

1. finding an off-line simulation that is faster than our on-line simulation of a tree machine by a *log-cost* RAM,
2. finding an optimal simulation of a pointer machine [12] by a *log-cost* RAM,
3. finding an optimal simulation of a unit-cost RAM by a *log-cost* RAM.

REFERENCES

[1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir, *A model for hierarchical memory*, in Proc. 19th Ann. ACM Symp. on Theory of Computing, 1987, pp. 305–314.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] S. A. Cook and R. A. Reckhow, *Time bounded random access machines*, J. Comput. System Sci., 7 (1973), pp. 354–375.

[4] Z. Galil, *Two fast simulations which imply some fast string matching and palindrome-recognition algorithms*, Inform. Process. Lett., 4 (1976), pp. 85–87.

[5] J. Katajainen, J. van Leeuwen, and M. Penttonen, *Fast simulation of Turing machines by random access machines*, SIAM J. Comput., 17 (1988), pp. 77–88.

[6] M. Li and P. M. B. Vitanyi, *Two decades of applied Kolmogorov complexity*, in Handbook of Theoretical Computer Science, Vol. A, J. van Leeuwen, ed., MIT Press, Cambridge, MA, and Elsevier, Amsterdam, 1990, pp. 187–254.

[7] M. C. Loui, *Optimal dynamic embedding of trees into arrays*, SIAM J. Comput., 12 (1983), pp. 463–472.

[8] ———, *Minimizing access pointers into trees and arrays*, J. Comput. System Sci., 28 (1984), pp. 359–378.

[9] ———, *The complexity of sorting on distributed systems*, Inform. and Control, 60 (1984), pp. 70–85.

[10] W. Paul and R. Reischuk, *On time versus space* II, J. Comput. System Sci., 22 (1981), pp. 312–327.

[11] K. R. Reischuk, *A fast implementation of a multidimensional storage into a tree storage*, Theoret. Comput. Sci., 19 (1982), pp. 253–266.

[12] A. Schönhage, *Storage modification machines*, SIAM J. Comput., 9 (1980), pp. 490–508.

[13] C. Slot and P. van Emde Boas, *The problem of space invariance for sequential machines*, Inform. and Comput., 77 (1988), pp. 93–122.

SIAM J. COMPUT.
Vol. 21, No. 5, pp. 972–999, October 1992

© 1992 Society for Industrial and Applied Mathematics
010

NEW RESULTS ON DYNAMIC PLANAR POINT LOCATION*

SIU WING CHENG† AND RAVI JANARDAN‡

Abstract. A point location scheme is presented for a dynamic planar subdivision whose underlying graph is only required to be connected. The operations supported include: reporting the name of the region containing a query point, inserting/deleting an edge, and inserting/deleting/moving a degree-2 vertex. The scheme uses $O(n)$ space, has a worst-case query time of $O(\log^2 n)$, and a worst-case update time of $O(\log n)$, where n is the number of vertices currently in the subdivision. Insertion (respectively, deletion) of an arbitrary k-edge chain inside a region can be performed in $O(k \log(n + k))$ (respectively, $O(k \log n)$) time in worst-case. The scheme outperforms the solutions given in works by Fries, Mehlhorn, and Naeher and by Overmars and also handles more general subdivisions than the schemes given in works by Preparata and Tamassia. The result is based on a new solution to a dynamic visibility problem for a set of line segments in the plane that are nonintersecting, except possibly at endpoints. The scheme is then extended to speed up the insertion/deletion of a k-edge monotone chain to $O(\log^2 n \log\log n + k)$ time (or $O(\log n \log\log n + k)$ time for an alternative model of input), but at the expense of increasing the other time bounds slightly. Additional results include a generalization to subdivisions consisting of algebraic segments of bounded degree and a persistent scheme that allows point location queries in the past and updates in the present.

Key words. computational geometry, dynamic data structure, planar subdivision, point location, priority search trees, trees of bounded balance

AMS(MOS) subject classifications. 68U05, 68Q25

1. Introduction. Given a planar subdivision, the point location problem is to determine the region in the subdivision that contains a given query point. Numerous results are known for the static version of this problem [1], [2], [3], [5], [7], [18], [22]. Recently, there have been a number of results for the dynamic version of the problem, where the subdivision can be changed by insertion/deletion of edges and vertices. Define a *connected subdivision* to be a planar subdivision whose underlying graph is connected. Let n be the number of vertices currently in the subdivision. Fries, Mehlhorn, and Naeher [4] reported a technique for connected subdivisions that achieves $O(n)$ space, $O(\log^2 n)$ query time, and $O(\log^4 n)$ amortized update time for inserting/deleting a vertex or an edge. If only insertions are considered, the amortized update time can be reduced to $O(\log^2 n)$ [10, pp. 135–143]. Later, Overmars [16] obtained another scheme for any subdivision in which the regions have a bounded number of edges. His scheme makes use of a dynamic segment tree and achieves $O(n \log n)$ space, $O(\log^2 n)$ query time, and $O(\log^2 n)$ worst-case time for inserting/deleting an edge or a vertex. (Overmars' scheme can also be used for connected subdivisions with no restriction on the region size.) Very recently, Preparata and Tamassia proposed efficient schemes for two special cases of the problem. One scheme is designed for convex subdivisions in which the n vertices lie on a fixed set of N horizontal lines [20]. It uses $O(N + n \log N)$ space and achieves $O(\log n + \log N)$ query time and $O(\log n \log N)$ worst-case time for inserting/deleting an edge or a vertex. The other scheme is designed for monotone subdivisions with no

*Received by the editors March 26, 1990; accepted for publication (in revised form) November 19, 1991. This research was partly supported by a grant-in-aid of research from the Graduate School of the University of Minnesota.

†Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455 (scheng@umn-cs.cs.umn.edu). Present address, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong.

‡Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455 (janardan@umn-cs.cs.umn.edu). The research of this author was also supported in part by National Science Foundation grant CCR-8808574.

restriction on the coordinates of vertices [19]. It uses $O(n)$ space and supports the following operations:

1. *Locate*(p): Given a query point p, report the name of the region containing p.

2. *Insertpoint*$(v, e; e_1, e_2)$: Split the edge $e = \{u, w\}$ into two edges $e_1 = \{u, v\}$ and $e_2 = \{v, w\}$ by inserting vertex v.

3. *Removepoint*$(v; e)$: Let v be a vertex of degree 2 whose incident edges, $e_1 = \{u, v\}$ and $e_2 = \{v, w\}$, are on the same straight line. Remove v, and replace e_1 and e_2 with edge $e = \{u, w\}$.

4. *Movepoint*$(v; x, y)$: Translate a degree-2 vertex v from its present location to point (x, y). (The operation is allowed only if the subdivision so obtained is monotone and topologically unchanged.)

5. *Insertchain*$(\gamma, v_1, v_2, r; r_1, r_2)$: Add the monotone chain $\gamma = v_1 w_1 \cdots w_{k-1} v_2$ inside region r, which is decomposed into regions r_1 and r_2, with r_1 and r_2 to the left and to the right of γ, respectively, when γ is directed from v_1 to v_2.

6. *Removechain*$(\gamma; r)$: Let γ be a monotone chain whose nonextreme vertices have degree 2. Remove γ and merge the regions r_1 and r_2 formerly on the two sides of γ into region r.

Monotonicity is required to be preserved after all the operations, and it is proved in [19] that the time complexities of the operations are (1) $O(\log^2 n)$, (2) $O(\log n)$, (3) $O(\log n)$, (4) $O(\log n)$, (5) $O(\log^2 n + k)$, and (6) $O(\log^2 n + k)$. All complexities are worst-case.

The main results in our paper are as follows.

First, we show that a very general class of subdivisions, namely, connected subdivisions, can be maintained efficiently in optimal $O(n)$ space. Connected subdivisions include the monotone subdivisions considered in [19] as a special case.

Second, in addition to supporting the above-mentioned operations *Locate*, *Insertpoint*, *Removepoint*, and *Movepoint* efficiently, our scheme also supports the following two operations.

- *Insert_edge*$(u, w, r; r_1, r_2)$. Insert the edge $\{u, w\}$ into region r, which contains u on its boundary. If w is also a vertex on the boundary of r, then the new edge $\{u, w\}$ splits the region r into two regions r_1 and r_2 such that r_1 and r_2 are to the left and right of $\{u, w\}$, respectively, when $\{u, w\}$ is directed from u to w. (If the subdivision is initially empty, then we simply create a subdivision consisting of $\{u, w\}$.)
- *Remove_edge*$(e; r)$. Remove an existing edge e. If e is on the boundary of two adjacent regions, then the two regions merge into a single region r.

The above repertory of dynamic operations is *complete* for connected subdivisions, i.e., a connected subdivision with n vertices can be assembled or disassembled with $O(n)$ operations from the repertory. Starting from an empty subdivision, we can assemble an n-vertex connected subdivision as follows. We first grow a spanning tree of the underlying graph by inserting the edges incrementally and then insert the rest of the edges arbitrarily. Conversely, we can disassemble an n-vertex connected subdivision by first deleting all the nontree edges with respect to a spanning tree of the underlying graph and then shrink the spanning tree remaining by deleting one edge at a time.

In our scheme, the query *Locate*(p) takes $O(\log^2 n)$ time in worst-case, while the time complexity of each of the update operations is $O(\log n)$ in worst-case. The update time for a single edge is an improvement on all solutions known previously [4], [10], [16], [20], [19], and the update time for a vertex matches the bound given in [19]. By repeated insertion (respectively, deletion) of edges, our scheme also allows insertion (respectively, deletion) of an arbitrary k-edge chain in $O(k \log(n + k))$ (respectively, $O(k \log n)$) time.

For $k = o(\log n)$, the time bound becomes $o(\log^2 n)$, which improves upon the result in [19], which applies to monotone chains only. We call this scheme Scheme I.

Third, we show how to modify Scheme I to speed up the insertion/deletion time of a k-edge monotone chain to $O(\log^2 n \log\log n + k)$, but at the expense of increasing the other time bounds to $O(\log^2 n \log\log n)$. We call this Scheme II. The insertion/deletion of monotone chains was explicitly addressed before only in [19] for monotone subdivisions. We also consider a less general model of input for Scheme II, where the updates specify pointers to the appropriate vertices and edges in the underlying graph rather than coordinates as assumed throughout the rest of the paper. For this less general model, the update time for a monotone chain is $O(\log n \log\log n + k)$, the query time $O(\log^2 n \log\log n)$, and the other update times $O(\log n \log\log n)$.

Table 1 compares our schemes for dynamic point location with previously known results.

We should emphasize that our schemes are mainly of theoretical interest because they involve rather complex manipulations of data structures, such as incremental rebuilding of auxiliary structures in a $BB(\alpha)$ tree [25] and global rebuilding [15], [17], which have large overhead. The design of a practical scheme that matches or improves our time bounds remains a challenging problem. In addition to the above-mentioned contributions, we also show how our scheme can be extended quite easily to accommodate subdivisions consisting of algebraic segments of bounded degree. Finally, we also present a persistent version of Scheme I, which allows point location queries in the past and updates in the present. No results were previously known for this problem, which was first posed by Sarnak and Tarjan [22].

The rest of the paper is organized as follows. Section 2 gives an overview of our basic approach. In §3, we solve a dynamic visibility problem that is closely related to the problem of planar point location. In §4.1, we show how to use the result in §3 to obtain Scheme I. We discuss Scheme II in §4.2. In §§4.3, we discuss the generalization to subdivisions consisting of algebraic segments of bounded degree. We present the persistent point location scheme in §5. Finally, we conclude in §6 with a discussion of some directions for future work.

2. An overview of the basic approach. We derive our dynamic point location schemes by solving the following dynamic visibility problem: we are to maintain a set of line segments in the plane that are nonintersecting (except possibly at endpoints) under insertions and deletions such that given a query point we can report efficiently the first segment that is hit when the point is moved horizontally to the right. If we store with each line segment (i.e., edge) of the subdivision the name of the region to its left, then the region containing a query point p can be found easily by answering a visibility query for p in the subdivision.

In [9] McCreight solves a restricted version of the above visibility problem, where the line segments are all vertical and the y-coordinates of their endpoints are in the range $[0, k-1]$ for some integer k. The solution in [9] uses a two-level tree structure: the outer tree B is a binary tree built by recursively bisecting the y-interval $[0, k-1]$. Each segment is stored at the highest node of B for which the segment intersects the node's bisector. The inner tree structure consists of two priority search trees (PSTs) at each node v, one each for the endpoints above and below v's bisector. A query is answered by binary searching down B, applying the PST query procedure *MinXinRectangle* [9] at each node visited to find the leftmost segment to the right of p at that node, and picking the leftmost of all such line segments found in the search down B. Updates are handled by simply inserting or deleting in the appropriate PST; no rebalancing of B is necessary

TABLE 1
Comparison of dynamic point location schemes. (All bounds given are "big–oh." "lg" *denotes logarithm to the base 2.) For lack of space, the function* $\lg n \lg\lg n$ *is abbreviated as* $L(n)$ *in the table.*

Scheme	Space	Query	Edge Update	Vertex Update	Insert/delete k-edge chain
Preparata et al. [19] Mon. subdiv.	n	$\lg^2 n$	$\lg^2 n$	$\lg n$	$\lg^2 n + k$ mon. chain only
Fries et al. [4] Conn. subdiv.	n	$\lg^2 n$	$\lg^4 n$ amortized	—	—
Overmars [16] Conn. subdiv.	$n \lg n$	$\lg^2 n$	$\lg^2 n$	$\lg^2 n$	$k \lg^2(n+k)$ (ins.) $k \lg^2 n$ (del.)
This paper Scheme I Conn. subdiv.	n	$\lg^2 n$	$\lg n$	$\lg n$	$k \lg(n+k)$ (ins.) $k \lg n$ (del.)
This paper Scheme II Conn. subdiv. (Pointer access)	n	$L(n) \lg n$	$L(n)$	$L(n)$	$L(n) + k$ mon. chain
					$k \cdot L(n+k)$ (ins.) $k \cdot L(n)$ (del.) any chain
This paper Scheme II Conn. subdiv. (Coord. access)	n	$L(n) \lg n$	$L(n) \lg n$	$L(n) \lg n$	$L(n) \lg n + k$ mon. chain
					$k \cdot L(n+k) \lg(n+k)$ (ins.) $k \cdot L(n) \lg n$ (del.) any chain

as the y-coordinates of the segment involved are in $[0, k-1]$ and so are already in B.

We use a similar approach for the general problem, with the following key differences.

First, the procedure *MinXinRectangle* can no longer be used if segments are not vertical because the segment corresponding to the endpoints that *MinXinRectangle* returns need not be to the right of p (as was the case for vertical segments in [9]). Instead, we devise a new PST querying procedure, *Find*, which searches the PST level by level and carefully prunes the search so that at most two nodes are examined at any level.

Second, in our case updates will require that B be rebalanced by means of rotations. These rotations will make it necessary to rebuild the PSTs at each affected node v. Doing the rebuilding all at once will be time-consuming. Instead, we spread the work over a sequence of future updates. We adapt a result due to Willard and Lueker [25], which essentially shows that if B is implemented as a $BB(\alpha)$ tree [14], [25], then it is possible to do only a constant number of rebuilding steps at v during each future update and still have the PSTs at v ready before v needs to be rotated again.

However, in order to realize our $O(\log n)$ update time for edges and vertices, we

must spend only $O(1)$ time per rebuilding step at v. Rebuilding the PSTs at v top-down, by repeated insertions, will be too expensive. Thus, a third key idea is to rebuild the PSTs bottom-up. Since a PST is essentially a heap, the total time spent in rebuilding it is linear in its size, and we can show that only $O(1)$ time is spent per rebuilding step.

Unfortunately, the bottom-up reconstruction gives rise to the following problem. While the reconstruction proceeds, we must still be able to answer queries and support updates on-line. While the PSTs at v are being rebuilt, we can still answer queries by suitably querying the old PSTs. Once the construction at v has been completed, the new PSTs take over. However, we cannot do updates on the PSTs under construction at v because they are being built bottom-up. Instead, we handle incoming insertions by inserting into a new PST (which is initially empty) at v. Deletions are even more problematic since they may specify a segment in the partially built PST, and we cannot delete efficiently from such a PST. Therefore, we perform a weak form of deletion where no rebalancing is done, and yet the query/update time does not deteriorate too much. We then apply the technique of global rebuilding [15], [17] to reconstruct the entire two-level structure before the original structure becomes too unbalanced.

In closing this section we mention that the idea of doing point location by means of visibility was first proposed by Overmars [16]. Overmars solved the problem by using a dynamic segment tree (implemented as a $BB(\alpha)$ tree). However, the fact that each segment gets split into $O(\log n)$ pieces in a segment tree leads to an $O(n \log n)$ space bound and an $O(\log^2 n)$ update bound in [16].

3. Solving a dynamic visibility problem. Let S be a set of line segments in the plane that are nonintersecting (except possibly at endpoints). Given a point p, we want to report the first segment in S hit by the horizontal ray, ray_p, that emanates from p to the right. If p happens to lie on one or more line segments, then one of them is arbitrarily chosen and reported. We present a solution for this problem that uses $O(n)$ space and supports the following operations.

1. *Locate_seg*(p): Return the first line segment hit by ray_p.
2. *Insert_seg*(s): Insert a line segment s into S assuming that s does not intersect any line segment currently in S.
3. *Delete_seg*(s): Delete a line segment s from S.

3.1. The data structure. For each node v in a binary tree T, we denote its parent, its left child, and its right child by *parent*(v), *left*(v), and *right*(v), respectively. We denote the subtree rooted at a node v by $T(v)$. We first present in §3.1.1 a preliminary structure with $O(\log n)$ *amortized* update time and then describe in §3.1.2 the necessary modifications to turn the amortized bound into worst-case.

3.1.1. The basics. We assume that we have started with an empty set and have performed a sequence of insertions and deletions to arrive at the current set S containing n line segments. We store S in a data structure $\Im$, which we describe below. The endpoints of a line segment s are denoted by $(x_l(s), y_l(s))$ and $(x_r(s), y_r(s))$, where $y_l(s) \geq y_r(s)$.

Each endpoint of a line segment in S is represented by a leaf in a $BB(\alpha)$ tree [14], [25], where $0 < \alpha < 1 - 1/\sqrt{2}$. We denote this tree by B. Each node v of B is assigned two numbers *rank*(v) and $\rho(v)$. The number *rank*(v) is one more than the total number of nodes in $T(v)$, including v. The number $\rho(v)$ is the ratio of *rank*(*left*(v)) to *rank*(v). A node v is *α-balanced* if and only if $\rho(v) \in [\alpha, 1-\alpha]$. As in [25], the range of balance at v is extended by choosing another value $\alpha' < \alpha$, so that v can be α'-balanced if not α-balanced. Formally, a value $\beta(v)$ is defined as follows:

$$\beta(v) = 2(\alpha - \alpha')^{-1}\max(0, \rho(v) - (1-\alpha), \alpha - \rho(v)).$$

Thus v is α'-balanced if and only if $\beta(v) \leq 2$, and α-balanced if and only if $\beta(v) = 0$.

Each leaf w stores a value $y(w)$, which is the y-coordinate of the endpoint represented by w. The leaves are ordered left to right by nonincreasing $y(\cdot)$ values, with ties broken arbitrarily. Every internal node v also stores a value $y(v)$, which is in the range $[y(w_2), y(w_1)]$, where w_1 (respectively, w_2) is the rightmost (respectively, leftmost) leaf in the subtree rooted at *left*(v) (respectively, *right*(v)). The value $y(v)$ is chosen at the time when v is first created and remains fixed throughout its lifetime. A horizontal line $Y(v) : y = y(v)$ is also defined for each node v. Let $S(v)$ consist of those line segments of S that intersect $Y(v)$ but not $Y(u)$ for any proper ancestor u of v. Note that each line segment of S is in $S(v)$ for exactly one v. Clearly, $\sum_u |S(u)|$ for all descendants u of v (including v) is at most $rank(v)/4$. For each line segment $s \in S(v)$, let $(x_s(v), y(v))$ be the point where $Y(v)$ intersects s.

At each internal node v, we store two *balanced priority search trees* (PSTs) [9], [10] $L(v)$ and $R(v)$. The underlying tree of $L(v)$ is chosen to be a leaf-oriented red-black tree. $L(v)$ stores the pairs $(x_s(v), y_l(s) - y(v))$ for all nonhorizontal line segments $s \in S(v)$. Each leaf of $L(v)$ stores an $x_s(v)$ value, and the $x_s(\cdot)$ values appear in nondecreasing order from left to right. With each node a of $L(v)$, we associate two fields: $a.y = y_l(s) - y(v)$ and $a.s = s$, where $a.y$ is the priority field [9], [10]. To be a priority search tree, the priority field of a node must be larger than or equal to those of its children. To this end, we store each nonhorizontal line segment s in $S(v)$ at exactly one ancestor a of the leaf storing $x_s(v)$ (i.e., we set $a.y$ to $y_l(s) - y(v)$ and $a.s$ to s) such that the above property is satisfied. $R(v)$ is defined similarly for the pairs $(x_s(v), y(v) - y_r(s))$. Each horizontal line segment s in $S(v)$ lies on $Y(v)$ and can be viewed as an interval $[x_s, y_s]$ on $Y(v)$. We represent each such interval $[x_s, y_s]$ by the pair (y_s, x_s) and store it in another PST denoted by $C(v)$.

During the insertion/deletion of segments, leaves will be inserted/deleted in B, and B may possibly become unbalanced. In this case, we try to perform a rotation to restore the balance. Figure 1(a) shows a single rotation and Fig. 1(b) shows a double rotation. We say that the nodes u and v in Fig. 1(a) or u, v, and w in Fig. 1(b) *actively participate* in the rotation. In both cases, we call the position of v before the rotation and the position of u after the rotation the *focus of the rotation*. Note that the previous values of $y(v)$, $y(u)$, and $y(w)$ are still valid after the rotation. Consider a node v that is not α-balanced but such that all the nodes in the subtree rooted at v are α'-balanced. Without loss of generality, suppose that $\rho(v) < \alpha$. It is shown in [25] that there exists a constant γ, $\alpha' < \gamma < 1 - \alpha'$, such that we can perform a left single rotation at v (Fig. 1(a)) if $\rho(right(v)) < \gamma$ or a left double rotation at v (Fig. 1(b)) if $\rho(right(v)) \geq \gamma$ to make the nodes that actively participated in the rotation α-balanced again. The symmetric case when $\rho(v) > 1 - \alpha$ can be handled similarly. In Fig. 1(a), the single rotation at v will cause those line segments in $S(v)$ that intersect $Y(u)$ to migrate to $S(u)$, thus making the PSTs at v and u obsolete (except for $C(v)$ and $C(u)$). Similarly, the double rotation in Fig. 1(b) will cause migration to $S(u)$ of those line segments in $S(v)$ and $S(w)$ that intersect $Y(u)$, thus making the PSTs at v, u, and w obsolete (except $C(v)$, $C(u)$, and $C(w)$). Therefore, we reconstruct, all at once, the PSTs $L(\cdot)$'s and $R(\cdot)$'s of the nodes that actively participate in that rotation. Otherwise, if $\beta(v)$ is greater than 2, then we rebuild the entire subtree $T(v)$ and also the PSTs $C(\cdot)$'s, $L(\cdot)$'s, and $R(\cdot)$'s in $T(v)$ all at once. If the PSTs are built by repeated insertions, then it can be proved that the amortized update time for inserting/deleting an edge is $O(\log^2 n)$. The proof is similar to the proof for Lemma 2.5 in [25]. The amortized update time is improvable to $O(\log n)$ if the PSTs are constructed by merging in a bottom-up fashion.

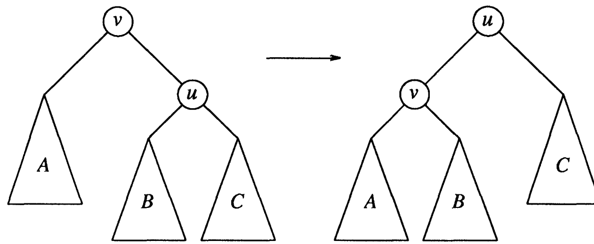


FIG. 1(a). *Single rotation.*

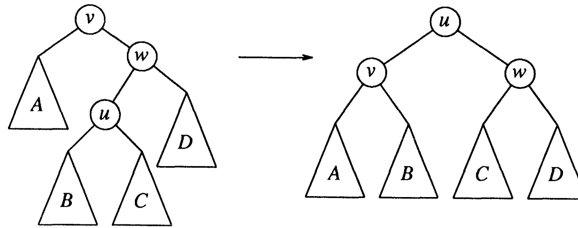


FIG. 1(b). *Double rotation.*

3.1.2. Refining the basic structure. To achieve a worst-case update time, we spread the bottom-up reconstruction of the PSTs over a sequence of future updates. Therefore, the time span in which the PSTs of some nodes of B are obsolete may spread over a sequence of updates. To signify that the PSTs of some nodes of B have become obsolete, we associate a flag, FLAG, with each node. For each node u, FLAG(u) is set to zero when the PSTs at u are up-to-date. It is reset to 1 after u has actively participated in a rotation. FLAG(u) may also take on values 2 and 3 subsequently, as explained in §3.3. A node u is *disunified* if FLAG(u) is nonzero; otherwise, u is *unified*. (A similar definition was given in [25]. However, in [25], the children of a disunified node were required to be unified in order to do the kinds of queries considered in [25]. We are able to drop this restriction here because our queries are different.) To identify the focus of a rotation and the node(s) that actively participated in that rotation, we associate another field MARK(u) with each node u. MARK(u) is *undefined* initially and is set as follows. After the single rotation in Fig. 1(a), we set MARK(u) to *singleL*. For the symmetric variant of Fig. 1(a), we set MARK(u) to *singleR*. After the double rotation in Fig. 1(b), or its symmetric variant, we set MARK(u) to *double*. FLAG$(\cdot)$ and MARK$(\cdot)$ will be useful in constructing the new PSTs for nodes that have actively participated in a rotation.

During the reconstruction, we still have to accommodate incoming insertions. Insertions cannot be performed in the PST being reconstructed because the reconstruction is bottom-up. Instead, we perform incoming insertions on a new (initially empty) PST. In fact, we let this new PST accommodate all the insertions at any time. Specifically, we keep two PSTs, $L_1(v)$ and $L_2(v)$, instead of a single $L(v)$. $L_2(v)$ is organized as described before, and insertions will be performed in it. The underlying tree of $L_1(v)$ is just a leaf-oriented balanced binary tree and not necessarily a red-black tree. For each nonhorizontal line segment s in $S(v)$, s is stored in either $L_1(v)$ or $L_2(v)$, but not both. $R(v)$ is similarly replaced by $R_1(v)$ and $R_2(v)$. Moreover, to facilitate the reconstruction, we also keep two doubly linked lists, *Leaf*$_1(v)$ and *Leaf*$_2(v)$, of line segments stored in $L_1(v)$ and $L_2(v)$, respectively. *Leaf*$_1(v)$ and *Leaf*$_2(v)$ are sorted by the $x_s(v)$ values,

and each leaf in $L_1(v)$ (respectively, $L_2(v)$) has a pointer to the corresponding entry in $Leaf_1(v)$ (respectively, $Leaf_2(v)$).

We must keep the old PSTs around for querying until the new ones are ready. Take v in Fig. 1(a) as an example. After the rotation, we rename $L_i(v)$ and $Leaf_i(v)$ as $L_{i+2}(v)$ and $Leaf_{i+2}(v)$, $i = 1, 2$. $L_i(v)$ and $Leaf_i(v)$, $i = 1, 2$, are then initialized to empty. Then we extract from $Leaf_3(v)$ and $Leaf_4(v)$ those line segments that now belong to $S(v)$, put them into $Leaf_1(v)$, and construct $L_1(v)$ from $Leaf_1(v)$ in a bottom-up fashion. Incoming insertions into $S(v)$ will be accommodated by $L_2(v)$ and $Leaf_2(v)$. Once $L_1(v)$ is completed, $L_3(v)$ and $L_4(v)$ are no longer needed and are discarded. The treatment of the $R_i(v)$'s is similar. The discussion of deletion will be postponed to §3.4.

Define the *near descendants* of a node to be itself, its children, and its grandchildren. After a rotation, the PSTs at the nodes that actively participated in the rotation will need to be updated. They should also be disallowed from actively participating in any further rotation before their PSTs are completely reconstructed. To this end, we define a node to be *eligible for rotation* if and only if all of its near descendants are unified; otherwise, it is *ineligible*. (Similar definitions were given in [25].)

3.2. Performing a visibility query. Let the query point be $p = (x_p, y_p)$. We first give a procedure *Find*$(p, r, y(v))$, which when applied at a node v of B to a PST $L_i(v)$ or $R_i(v)$ with root r returns the first segment, $s^\star$, in the PST that is hit by ray_p. If there is no such segment $s^\star$, then the line $x = \infty$ is returned. (Note that if the line segments in the $L_i(v)$'s and $R_i(v)$'s were all vertical, then the procedure *MinXinRectangle* given in [9] could be used. However, *MinXinRectangle* will not work in our case because a comparison between x_p and $x_s(v)$, for some line segment s, cannot tell whether p is to the right or to the left of s (see Fig. 2(a)). Nor is it true that a comparison between p and the supporting line of s can guide the search (see Fig. 2(b)).)

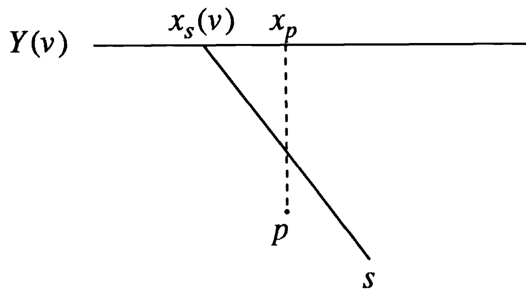


FIG. 2(a). $x_p > x_s(v)$, *but p is to the left of s.*

Find explores the PST level by level from r, always maintaining the invariant that at most two nodes are examined at any level, and $s^\star$ (if it exists) is in the subtree rooted at one of these nodes. To facilitate this, the two nodes to be examined are maintained in a queue, Q. Q is initialized to contain r and *Find* terminates when Q becomes empty. A variable, *answer*, is initialized to the line $x = \infty$ and at termination contains the answer to *Find*. If Q is not empty, then a node, a, is deleted from the front of Q. Let $\Delta = |y_p - y(v)|$. If $a.y \geq \Delta$ and p is to the left of $a.s$, then *answer* is reset to the leftmost of *answer* and $a.s$. Then Q is updated according to the following cases: (i) if $\Delta >$ *left*$(a).y$ and $\Delta >$ *right*$(a).y$ (Fig. 3(a)), then the subtrees $T($*left*$(a))$ and $T($*right*$(a))$ cannot contain $s^\star$ since the PST is a maxheap. So Q is left unchanged; (ii) if $\Delta \leq$ *left*$(a).y$ and p is to the left of *left*$(a).s$ (Fig. 3(b)), then $s^\star$ cannot be in $T($*right*$(a))$ since the segments are nonintersecting. However, $s^\star$ can be in $T($*left*$(a))$ and so we insert *left*(a) at the end of

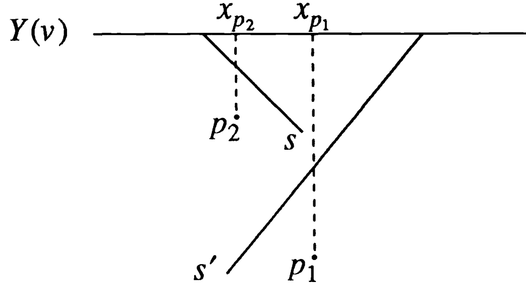


FIG. 2(b). *Both p_1 and p_2 are to the left of the supporting line of s. But, no segment to the left of s can be the answer for p_1, while no segment to the right of s can be the answer for p_2.*

Q. Similarly, for the symmetric case. (iii) If $\Delta \leq left(a).y$ and p is to the right of $left(a).s$ and $\Delta > right(a).y$ (Fig. 3(c)), then $s^\star$ cannot be in $T(right(a))$ but can be in $T(left(a))$. So we insert $left(a)$ into Q. Similarly for the symmetric case. (iv) If $\Delta \leq left(a).y$ and $\Delta \leq right(a).y$ and p lies between $left(a).s$ and $right(a).s$ (Fig. 3(d)), then $s^\star$ cannot be to the left of $left(a).s$ or to the right of $right(a).s$. Thus, $s^\star$ can only be in $T(left(a))$ or in $T(right(a))$ and so we empty Q and then insert $left(a)$ and $right(a)$ into it.

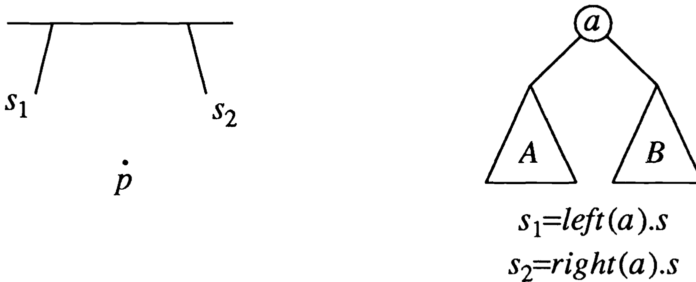


FIG. 3(a). *No need to search A and B.*

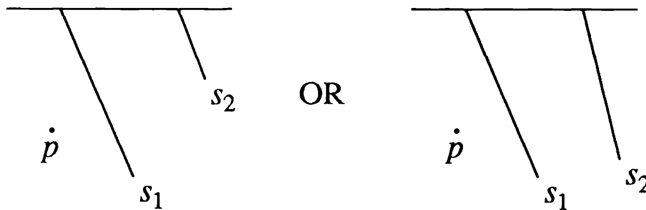


FIG. 3(b). *No need to search candidate subtrees that are to the right of A.*

LEMMA 1. *$Find(p, r, y(v))$ correctly locates in $O(\log n)$ time the first line segment hit by ray_p among those line segments stored in $T(r)$, where $T(r)$ is a* PST *associated with v and n is the number of leaves in $T(r)$.*

Proof. The cases listed exhaust all possibilities. Furthermore, it is clear from the above discussion that *Find* maintains the invariant throughout its execution and is hence correct. Since *Find* spends $O(1)$ time per level and the PST has height $O(\log n)$, its running time is $O(\log n)$. □

$Locate_seg(p)$ does a binary search down B using y_p until the search either runs

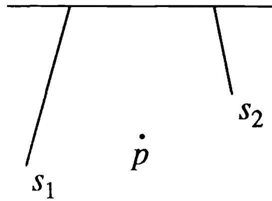


FIG. 3(c). *No need to search B.*

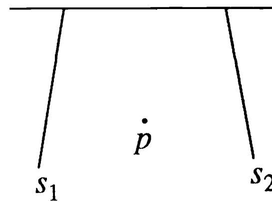


FIG. 3(d). *Restrict the search to A and B.*

off B at a leaf or reaches a node u such that $y_p = y(u)$. At each node v visited we do the following: If $y_p = y(v)$, then we query $C(v)$ as described in [9], [10]. If a segment of $C(v)$ is found to contain p, then $Locate_seg(P)$ terminates. Otherwise, we proceed as follows: Suppose that v is unified. If $y_p \geq y(v)$ (respectively, $y_p < y(v)$), then we call *Find* on $L_1(v)$ and $L_2(v)$ (respectively, $R_1(v)$ and $R_2(v)$). On the other hand, suppose that v is disunified. If $y_p \geq y(v)$, then we call *Find* on $L_2(v)$, $L_3(v)$, and $L_4(v)$. Moreover, since some of the segments in $S(v)$ may be stored in some child of v, we also do the following. If MARK(v) equals *singleL*, then if $y_p \geq y(left(v))$ (respectively, $y_p < y(left(v))$), then we also call *Find* on $L_3(left(v))$ and $L_4(left(v))$ (respectively, $R_3(left(v))$ and $R_4(left(v))$) because some of the segments stored there may actually belong to $S(v)$ now. If MARK(v) = *singleR*, then we call *Find* on $L_3(right(v))$ and $L_4(right(v))$ (since we have $y_p \geq y(right(v))$ also). If MARK(v) = *double*, then we query the PSTs at both *left*(v) and *right*(v), following the steps above for *singleL* and *singleR*, respectively. The remaining case of v being disunified and $y_p < y(v)$ can be handled symmetrically as above.

When $Locate_seg(P)$ terminates, if a segment has been found to contain p, then we report that segment. Otherwise, among the segments found by the *Find*s, we report the leftmost.

LEMMA 2. *Suppose a set, S, of line segments in the plane that are nonintersecting (except possibly at endpoints) is stored in $\Im$. Then given a query point p, $Locate_seg(p)$ correctly determines the first line segment hit by ray_p in $O(\log^2 n)$ time.*

Proof. Let s^* be the first line segment hit by ray_p. Let w_1 and w_2 be the leftmost and rightmost leaves in B such that $[y(w_1), y(w_2)]$ coincides with the projection of s on the y-axis. Then we have the inequality $y(w_1) \leq y(p) \leq y(w_2)$. This inequality and the choice of w_1 and w_2 imply that the lowest common ancestor v of w_1 and w_2 must be visited during the binary search down B. Since $y(w_1) \leq y(v) \leq y(w_2)$, s^* must intersect $Y(v)$ and $s^* \in S(u)$ for some ancestor of v (u may possibly be v itself). By the querying strategy described, if u is unified, we query some PSTS at u and locate the leftmost line segment in $S(u)$ hit by ray_p. Otherwise, we also query some of the PSTs at a child u' of u if MARK(u) = *singleL* or *singleR* or both children u' and u'' of u if MARK(u) = *double*. Since an ineligible node is not allowed to actively participate in a rotation, s^* must be

stored either at u or possibly at u' (respectively, u' or u'') if MARK(u) = *singleL* or *singleR* (respectively, if MARK(u) = *double*). Therefore, s^* must have been reported after we finished the querying at v. This proves the correctness of *Locate_seg*(p). By Lemma 1, we spend $O(\log n)$ time at each node visited. Since B is α'-balanced, at most $O(\log n)$ nodes are visited, and hence *Locate_seg*(p) takes $O(\log^2 n)$ time. □

3.3. Performing *Insert_seg*(s).

3.3.1. The algorithm. We follow the approach introduced by Willard and Lueker [25] to insert s efficiently into $\mathfrak{T}$. In procedure *Insert_seg*, we first invoke *Insert_leaf* to insert two leaves corresponding to the horizontal lines through the two endpoints of s into B. Then we locate the node v such that $s \in S(v)$ and insert s into $C(v)$ or into $L_2(v)$, $R_2(v)$, and $Leaf_2(v)$. In procedure *Insert_leaf*, we have to handle the possibility that B becomes unbalanced after the insertion of a leaf. If a node v on the path from root to the newly inserted leaf becomes unbalanced, then we will perform a rotation at v if $\beta(v)$ is in the range $(1,2]$ and v is eligible. Otherwise, if $\beta(v) > 2$, then we do a brute force reconstruction of an entire subtree rooted at v. Hence, it is guaranteed that B is α'-balanced at all times. We also perform a constant number of steps in the incremental reconstruction of PSTs for the disunified near descendants of *ineligible* nodes on the path from root to the newly inserted leaf. For each disunified node u, the incremental reconstruction is divided into three phases that are carried out by the procedures *Fixup*1, *Fixup*2, and *Fixup*3, respectively, which are given below. *Fixup*1 collects the line segments in $S(u)$ in several lists, each sorted by the $x_s(u)$'s values. *Fixup*2 merges these lists into $Leaf_1(u)$. *Fixup*3 constructs the new $L_1(u)$ and $R_1(u)$ from $Leaf_1(u)$ in a bottom-up fashion.

```
Procedure Insert_seg(s)
    call Insert_leaf(y_l(s))
    call Insert_leaf(y_r(s))
    locate the highest v in B such that s intersects Y(v)
    if s lies on Y(v) then
        insert s into C(v)
    else
        insert s into L2(v) and R2(v)
        insert s into Leaf2(v); this can be done in O(1) time as the
            proper position for s can be determined during the searching in L2(v)
    end if
end Insert_seg

Procedure Insert_leaf(y)
    binary search down B using y
    let w be the leaf at which the search terminates and
        let P be the search path to w
    replace w by a new internal node u
    create a new leaf w', store y in w', and make w and w'
        the appropriate children of u
    choose a value between y(w) and y(w') for y(u)
    if any node v on P has β(v) > 2 then
        PANIC: Let v be the highest node on P with β(v) > 2.
        Rebuild the subtree rooted at v into a BB(α) tree and rebuild
```

```
        C(·)'s, L_i(·)'s, R_i(·)'s, and Leaf_i(·)'s, i = 1,2. (L_i(·)'s, R_i(·)'s, and
        Leaf_i(v)'s, i = 3,4 are not needed).
    end if
    for each node v on P do
        if v is ineligible then
            (* In what follows, let c be a constant whose value will be
            chosen later *)
            for i = 1 to 3 do
                call Fixupi(u) for a disunified near descendant u of v such that
                FLAG(u) = i until no such u exists or c calls have been made
            end for
        end if
        if v is eligible and β(v) > 1 then
            ROTATE:
                rebalance v by a single or double rotation
                set MARK(v') appropriately, where v' is the node replacing v
                for each node u that has actively participated in the rotation do
                    set FLAG(u) to 1
                    rename L_i(u), R_i(u), and Leaf_i(u) as L_{i+2}(u), R_{i+2}(u),
                        and Leaf_{i+2}(u), i = 1,2
                    set L_i(u), R_i(u), Leaf_i(u) to empty, i = 1,2
                    set Out(u, left(u)), Out(u, right(u)), In_3(u), and In_4(u) to empty
                    (* Out(·,·) and In_j(·) will be used in Fixup1 and Fixup2 *)
                end for
        end if
    end for
end Insert_leaf
```

*Fixup*1(u), where u is a disunified node, works as follows. Using MARK(u), we first determine the focus of rotation and set u to it. For each child v of u that actively participated in the rotation, a call to *Fixup*1(u) identifies one line segment from $Leaf_3(v) \cup Leaf_4(v)$ that should migrate to u, as follows. We delete the leftmost line segment s in $Leaf_3(v) \cup Leaf_4(v)$, and if s intersects $Y(u)$, then we append s to a list $Out(u, v)$ associated with u; otherwise, we append s to one of the lists, $In_3(v)$ or $In_4(v)$, depending on whether s came from $Leaf_3(v)$ or $Leaf_4(v)$. These lists will be used by *Fixup*2. After $Leaf_3(v)$ and $Leaf_4(v)$ have been exhausted for each child v of u that actively participated in the rotation, we set FLAG(u) and FLAG(v) to 2 to signal the start of the second phase in the incremental reconstruction.

```
Procedure Fixup1(u)
    if MARK(u) = undefined then u := parent(u) end if
    case MARK(u) of
        singleL : v_1 := left(u)
        singleR : v_1 := right(u)
        double : v_1 := left(u); v_2 := right(u)
    end case
    if MARK(u) = double then
        for i := 1 to 2 do
            if Leaf_3(v_i) ≠ ∅ or Leaf_4(v_i) ≠ ∅ then
```

```
                choose s such that x_s(v_i) is least in Leaf_3(v_i) ∪ Leaf_4(v_i)
                let s be picked from Leaf_j(v_i), j = 3 or 4
                delete s from Leaf_j(v_i)
                if s intersects Y(u) then
                    append s to Out(u, v_i)
                else
                    append s to In_j(v_i)
                end if
            end if
        end for
        if Leaf_3(v_1) = Leaf_4(v_1) = Leaf_3(v_2) = Leaf_4(v_2) = ∅ then
            set FLAG(u), FLAG(v_1), and FLAG(v_2) to 2
        end if
    else
        if Leaf_3(v_1) ≠ ∅ or Leaf_4(v_1) ≠ ∅ then
            choose s such that x_s(v_1) is least in Leaf_3(v_1) ∪ Leaf_4(v_1)
            let s be picked from Leaf_j(v_1), j = 3 or 4
            delete s from Leaf_j(v_1)
            if s intersects Y(u) then
                append s to Out(u, v_1)
            else
                append s to In_j(v_1)
            end if
        else
            set FLAG(u) and FLAG(v_1) to 2
        end if
    end if
end Fixup1
```

Each call to *Fixup*$2(u)$, where u is disunified, performs one step in the incremental reconstruction of the new leaf list $\mathit{Leaf}_1(u)$. If MARK(u) is *undefined*, then the desired list $\mathit{Leaf}_1(u)$ is the merge of $\mathit{In}_3(u)$ and $\mathit{In}_4(u)$. Thus we move the leftmost line segment from $\mathit{In}_3(u) \cup \mathit{In}_4(u)$ to $\mathit{Leaf}_1(u)$. Otherwise, the desired $\mathit{Leaf}_1(u)$ is the merge of $\mathit{Leaf}_3(u)$, $\mathit{Leaf}_4(u)$, $\mathit{Out}(u, \mathit{left}(u))$, and $\mathit{Out}(u, \mathit{right}(u))$. We move the leftmost line segment from these four lists to $\mathit{Leaf}_1(u)$. Note that this will involve computing the intersection with $Y(u)$ of the first line segment in each $\mathit{Out}(u, \cdot)$ list. Once the relevant lists of candidate line segments at u have been exhausted, we set FLAG(u) to 3 to signal the start of the third phase.

```
Procedure Fixup2(u)
    if MARK(u) = undefined then
        if In_3(u) or In_4(u) is not empty then
            delete the segment s with the least x_s(u) value in the lists
                In_3(u) and In_4(u)
            append s to Leaf_1(u)
        else
            set FLAG(u) to 3
        end if
    else
```

```
      if Leaf_3(u) or Leaf_4(u) or Out(u, left(u)) or Out(u, right(u)) not empty then
         delete the segment s with the least x_s(u) value in the lists
            Leaf_3(u), Leaf_4(u), Out(u, left(u)), and Out(u, right(u))
         append s to Leaf_1(u)
      else
         set FLAG(u) to 3
      end if
   end if
end Fixup2
```

Procedure *Fixup*3(u), where u is disunified, performs one step in the bottom-up construction of each of $L_1(u)$ and $R_1(u)$ using the list *Leaf*$_1(u)$ built in the previous phase.

Procedure *Fixup*3(u)

Perform one step of the following construction of $L_1(u)$ and $R_1(u)$. Build a binary search tree bottom-up from the ordered list of line segments in *Leaf*$_1(u)$. Then fill in the nodes bottom-up, using a knockout tournament much like the heap construction in Heapsort. Specifically, set the fields $a.s$ and $a.y$ of a leaf a of $L_1(u)$ (respectively, $R_1(u)$) that stores $x_s(u)$ to s and $y_l(s) - y(u)$ (respectively, $y(u) - y_r(s)$). Once the fields of the children of a node a' have been computed, copy the fields of the child of a' with the larger priority field to a'. Then compute the fields of the vacated child from that of its children in a similar fashion; once a leaf is reached, set its priority field to $-\infty$. When the construction is completed, set FLAG(u) to 0 and MARK(u) to *undefined*. Also remove $L_i(v)$ and $R_i(v)$, $i = 3, 4$.

end *Fixup*3

3.3.2. Analysis of *Insert_seg*. Except for the block PANIC in *Insert_leaf*, it is clear that each call to *Insert_leaf* takes $O(\log n)$ time plus the time for c calls each to *Fixup*1, *Fixup*2, and *Fixup*3, which is $O(\log n)$ as each call to *Fixup*1, *Fixup*2, and *Fixup*3 takes constant time. By adapting the proof for a similar result due to Willard and Lueker [25], it can be shown that PANIC will only be invoked at a node of constant rank and thus takes $O(1)$ time. In [25], range restriction capability is added to an arbitrary dynamic data structure as follows. The objects are sorted by some value associated with them and are stored at the leaves of a range tree organized as a $BB(\alpha)$ tree. Each node in the range tree is augmented with an instance of that dynamic data structure consisting of all the objects stored at the leaves descending from that node. The range tree is rebalanced after an insertion mainly by rotations, but it is necessary to rebuild an entire subtree from time to time. Willard and Lueker [25] prove that this rebuilding step takes $O(1)$ time.

In the following, we describe the underlying intuition for proving that PANIC takes $O(1)$ time.

We first make the following definitions (which parallel those in [25]). An *update* is a complete execution of *Insert_leaf*. Suppose that t and t' are points in time between updates (not during an update). Let $\min(v, t, t')$ (respectively, $\max(v, t, t')$) denote the minimum (respectively, maximum) rank of v at any time from t to t', excepting times when an update execution was partially finished. A node v is said to be *involved* in an update if a leaf is inserted into the subtree rooted at v. We call v the *focus* of PANIC if PANIC is applied to v.

If PANIC is invoked at a node v, it is because v has been ineligible for some time, thus prohibiting rotation and allowing $\beta(v)$ to gradually increase to greater than two.

Recall that for each update involving a node v, we perform a constant number of steps of incremental reconstruction for the disunified near descendants of v. Our goal is to prove that if *rank*(v) is bigger than some constant to be specified later, then there must be sufficient number of updates involving v to increase $\beta(v)$ from less than or equal to one to close to two such that the incremental reconstruction at the disunified near descendants of v should have finished by then. Thus, v becomes eligible before it is necessary to invoke PANIC at v. It follows that PANIC is invoked only at nodes whose rank is bounded from above by some constant.

Intuitively, given a certain number of updates involving v, the larger the value of *rank*(v), the smaller will be the ratio of the number of these updates to *rank*(v). Therefore, a smaller increase in $\beta(v)$ would be expected for a larger *rank*(v). On the other hand, $\beta(v)$ would be expected to increase with the number updates involving v. This is formalized in Lemma 3 below (Lemma 4.2 in [25]). Lemma 3 implies that in order to increase $\beta(v)$ from a value less than or equal to one to a value close to but less than two (i.e., just before it becomes necessary to apply PANIC to v), there must have been $\Omega(rank(v))$ updates involving v.

LEMMA 3 (Lemma 4.2 in [25]). *There exists a constant ξ such that if i_v updates involving node v occur between t_1 and t_2, then $\beta(v)$ will increase by no more than $\xi i_v / \min(v, t_1, t_2)$ during this period (including $\beta(v)$'s value during the execution of the updates). Furthermore, if no update other than possibly the last in this period triggers a rotation or* PANIC *that changes* rank(v), *then the increase in $\beta(v)$ will also be bounded by $\xi i_v / \max(v, t_1, t_2)$.*

Furthermore, as stated in Lemma 4 below (Lemma 4.4 in [25]), within the above $\Omega(rank(v))$ updates there cannot be more than a constant number of times when a near descendant of v is subject to a rotation or PANIC. This can be explained by the fact that *rank*(v) and the rank of a near descendant of v differ only by a constant factor. Lemma 4 implies that there are $\Omega(rank(v))$ consecutive updates within the above sequence of i_v updates such that no rotation or PANIC is applied to a near descendant of v.

LEMMA 4 (Lemma 4.4 in [25]). *Suppose that between times t_1 and t_2, node v never actively participates in a rotation or* PANIC. *Suppose further that at most $\lceil \min(v, t_1, t_2) \cdot (\alpha')^2 / \xi \rceil$ updates involving v occur between times t_1 and t_2. Then there can be no more than ten occasions when a child or grandchild of v is the focus of a rotation or* PANIC *during this time interval.*

Finally, in Lemma 5 below, we prove that at most $O(rank(v))$ steps of incremental reconstruction are needed to render v eligible, provided that the incremental reconstruction is uninterrupted by any rotation or invocation of PANIC at some near descendant of v. Lemmas 3–5 together imply Lemma 6, the desired result, i.e., PANIC is only applied to nodes of some constant rank. A formal proof for Lemma 6 can be adapted from the proof of an analogous result, Lemma 4.5, in [25].

LEMMA 5. *Let $rank(\cdot)_t$ denote the rank of a node at time t. There exists a constant $\lambda > 0$ such that if at time t a node v is ineligible, then a total of no more than $\lambda \cdot rank(v)_t$ calls to Fixup1, Fixup2, and Fixup3 will be needed for v's disunified near descendants before v becomes eligible, assuming that no near descendant of v is the focus of a rotation or* PANIC *during these calls.*

Proof. Let u be a disunified near descendant of v. By the assumption of the lemma, we know that once u becomes unified, it remains so during this incremental reconstruction. Therefore, we shall construct a $L_1(u)$ and a $R_1(u)$ for u exactly once. Note that *rank*$(\cdot)$ is nondecreasing with respect to time (because we never remove a leaf during deletion as we shall see later). Thus, the sum of the sizes of the relevant $Leaf_3(\cdot)$'s and $Leaf_4(\cdot)$'s that need to be examined for incremental reconstruction at u after time t is

bounded by $rank(parent(u))_t/4 \leq rank(u)_t/4\alpha'$. Hence, *Fixup*1$(u)$ needs to be called at most $rank(u)_t/4\alpha'$ times before FLAG(u) becomes 2. Then *Fixup*2(u) needs to be called at most $|S(u)| \leq rank(u)_t/4$ times before FLAG(u) becomes 3. *Fixup*3(u) resembles the bottom-up construction of a heap and requires at most $\lambda' \cdot rank(u)_t/4$ steps for some fixed $\lambda' > 0$. Thus no more than $\lambda'' \cdot rank(u)_t$ calls of *Fixup*1(u), *Fixup*2(u), and *Fixup*3(u) are needed before u becomes unified, where $\lambda'' = (1 + \alpha' + \lambda'\alpha')/4\alpha'$. Summing up for all the near descendants of v, we get the bound $3\lambda'' \cdot rank(v)_t = \lambda \cdot rank(v)_t$, where $\lambda = 3\lambda''$. □

LEMMA 6. *There exists a positive constant c for Insert_leaf such that if* PANIC *is applied to a subtree $T(v)$ of B, then $rank(v)$ does not exceed* $\max(3\xi, 9)$, *where ξ is the constant in Lemmas* 3 *and* 4.

We may now conclude that the PANIC block takes $O(1)$ time. Then the $O(\log n)$ bound for the insertion time follows immediately from the discussion at the beginning of this section.

LEMMA 7. *Insert_seg(s) takes $O(\log n)$ worst-case time.*

COROLLARY 1. *The data structure $\Im$ for a set of n line segments can be built in time* $P(n) = O(n \log n)$.

Proof. Starting with $\Im$ empty, repeatedly insert each segment using *Insert_seg*. □

3.4. Handling deletions. In doing *Delete_seg*(s), we have to delete s from some $L_i(v)$'s. If s is stored in $L_1(v)$ or $L_3(v)$, then the deletion becomes problematic. First, the underlying trees of $L_1(v)$ and $L_3(v)$ may not be red-black trees due to the bottom-up construction in *Fixup*3. Therefore, a deletion may cause more than $O(1)$ rotations, and hence the time required may be more than $O(\log n)$. Second, $L_1(v)$ may still be under construction, and, therefore, a deletion will interfere with the incremental reconstruction. Similarly, for deleting the item representing s in $R_i(v)$'s, $i = 1, 3$. To avoid this problem, we define an operation, *Weak_Delete*(s), which deletes all the occurrences of s in $\Im$ but does no structural change to $L_i(v)$ and $R_i(v)$, $i = 1, 3$. *Weak_Delete*(s) is a *weak deletion* [15], [17], i.e., it does not rebalance the data structure but it still guarantees the correctness of subsequent queries. Furthermore, after δn deletions for some $0 < \delta < 1$, the insertion time (respectively, query time, weak deletion time) of the "unbalanced" structure are still bounded by k_δ times the insertion time (respectively, query time, weak deletion time) of a corresponding balanced structure on the remaining $(1-\delta)n$ elements, where k_δ is a constant depending only on δ.

Procedure *Weak_Delete*(s)
We assume the existence of a global dictionary which allows us
to access the locations where s is stored in $O(\log n)$ time
for each node v storing s do
(* there are at most two such nodes *)
if s is in $L_i(v)$ and $R_i(v)$, $i = 1, 3$, then
set the priority fields of nodes storing s to $-\infty$
demote $(x_s(v), -\infty)$ to the leaves for s in $L_i(v)$ and $R_i(v)$
note that demotion can be done even when $L_1(v)$ or $R_1(v)$
is under construction
end if
if s is in $C(v)$ or in $L_i(v)$ and $R_i(v)$, i=2 or 4, then
do a PST deletion of s in $C(v)$ or in $L_i(v)$ and $R_i(v)$.
end if
if s is in $Leaf_i(v)$ for some i, or s is in $Out(v, u)$, for $u = left(v)$ or $right(v)$,

or s is in $In_j(v)$, for some j, then
delete s from those doubly linked lists in $O(1)$ time
end if
end *Weak_Delete*

LEMMA 8. *The operation Weak_Delete(s) on $\Im$ is a weak deletion and takes $O(\log n)$ time, where n is the number of line segments currently stored in $\Im$.*

We then apply global rebuilding [15], [17] to convert the weak deletions to strong ones, i.e., to *Delete_seg*(s), without affecting the insertion time and query time. Essentially, the construction of a fresh copy of $\Im$ to replace the old "unbalanced" one is triggered from time to time, and the work is spread over sufficiently many future updates so that each update can still be performed efficiently.

LEMMA 9. *A line segment can be strongly deleted from $\Im$ in $O(\log n)$ worst-case time.*

Hence, we have obtained the desired structure for the dynamic visibility problem.

THEOREM 1. *Using $\Im$, we can dynamically maintain a set of line segments in the plane that are nonintersecting (except possibly at endpoints) in $O(\log n)$ worst-case update time and using $O(n)$ space, so that given a query point p, the first line segment hit by ray_p can be located in $O(\log^2 n)$ time.*

4. Dynamic point location. We first give an overview of our structure for dynamic planar point location, which consists of two modules. One is the main structure for supporting queries and updates. It is essentially the data structure $\Im$ discussed in §3 subject to some modifications. The other module is an interface between the main structure and the user. It is an adjacency list representation, AL, of the underlying graph of the planar subdivision. For each vertex, v, we store its coordinates, and for each of its incident edges we store pointers to the other end vertex in AL and to the occurrences of that edge in the main structure. The edge list of each vertex is organized as a balanced search tree sorted by the angles between the incident edges and a horizontal line through that vertex. Thus, a new edge $\{v, w\}$ can be inserted in $O(\log n)$ time, and its two adjacent edges incident to v (or w) can also be located in $O(\log n)$ time. Moreover, insertion/deletion of a k-edge chain in AL takes $O(\log n + k)$ time since intermediate vertices on the chain have degree two.

For some applications that also maintain the underlying graph of the subdivision, the structure AL suffices because the input can then be specified as pointers to vertices and/or edges in AL. We can then access the occurrences of the corresponding vertices and/or edges in the main structure by following certain pointers. Otherwise, if a vertex is only specified by its coordinates and an edge by the pair of coordinates of its end vertices, then we need two dictionaries, D_V and D_E. D_V is searched using the coordinates of a vertex as key and provides a pointer to the corresponding node in AL. D_E works similarly for edges. Clearly, both D_V and D_E use $O(n)$ space and can be maintained in $O(\log n)$ time per update.

In §4.1, we shall present Scheme I that supports point location query, insertion/deletion of an edge, and insertion/deletion/moving of a degree-2 vertex. Scheme I has the same performance in both models of input mentioned above. We then present Scheme II in §4.2, which is an enhancement of Scheme I. Scheme II allows more efficient updates of chains that are monotone. (Insertion/deletion of a single edge is now treated as an insertion/deletion of a monotone chain of length 1.) However, the other time bounds increase slightly. The performance of Scheme II is better if the input is specified as pointers to vertices and/or edges in AL. In §4.3, we generalize our scheme to subdivisions consisting of algebraic segments of bounded degree. Lastly, we extend Scheme I

to obtain a persistent planar point location scheme in §5.

4.1. Scheme I. Given a connected planar subdivision with n vertices, we store all the edges of the subdivision as well as a vertical line at $x = \infty$ in $\Im$. Given a query point p, we use *Locate_seg*(p) in §3 to find the first edge that is hit by ray_p. Let the edge be e. If e is the vertical line at $x = \infty$, then p must be in the exterior region of the subdivision. Suppose that e is an edge in the subdivision. If e contains p, then we just report e as the answer. Otherwise, the region containing p is the region immediately to the left of e. In order to report the name of this region and to allow efficient update of regions, we use a concatenable queue augmented with parent pointers to store the boundary of each region. Specifically, let $\{v_1, v_2, \cdots, v_l\}$ be a clockwise ordering of vertices on the boundary of a region r. We store the edges $\{v_1, v_2\}, \cdots, \{v_{l-1}, v_l\}, \{v_l, v_1\}$ in this order at the leaves of a concatenable queue. The name r is stored at the root. Since each edge e in $\Im$ belongs to the boundary of two regions, we keep two copies of e in two concatenable queues. We also store at e two pointers to these copies. We assume that only coordinates are given in the input, and we obtain access to the occurrences of the vertices and/or edges in $\Im$ and the concatenable queues using D_V, D_E and AL. Updating these three structures takes $O(\log n)$ time trivially, and so we will omit further discussion of this in what follows.

Consider *Locate*(p). Let e be the edge reported by *Locate_seg*(p). We follow the pointer at e to the leaf of the concatenable queue that represents the boundary of the region to the left of e. Then we follow parent pointers to go to the root of the queue and report the name stored there.

Consider *Insert_edge*$(u, w, r; r_1, r_2)$. We first insert the edge $\{u, w\}$ into $\Im$ in $O(\log n)$ time. We then create a node, arc_1, for the arc directed from u to w and another node, arc_2, for the arc directed from w to u. We use the dictionaries to locate in $O(\log n)$ time the two leaves storing the two boundary edges of r that are incident to u. If $\{u, w\}$ does not split r into two regions, then $\{u, w\}$ extends the boundary of r. We split the concatenable queue for r in between the two edges incident with u into two pieces, q_3 and q_4, where q_4 may be empty. The split is done bottom-up by following pointers from leaf to root. Then we merge q_3, arc_1, $arcv_2$, and q_4 in this order to form the new concatenable queue for r. Suppose that $\{u, w\}$ splits r into two regions, r_1 and r_2. Then we split the concatenable queue for r as above into three pieces, q_3, q_4, and q_5, where q_5 may be empty. Without loss of generality, let q_3, arc_1, and q_5 form the boundary of r_1. We merge q_3, arc_1, and q_5 in this order to obtain the concatenable queue for r_1. Similarly, we merge arc_2 and q_4 in this order to obtain the concatenable queue for r_2. Finally, we store the names r_1 and r_2 at the respective roots of the concatenable queues obtained. This step involves a constant number of splittings and mergings of concatenable queues and thus runs in $O(\log n)$ time. *Remove_edge*(e, r) is simply the reverse of the above.

Intuitively, we can implement *Insertpoint*$(v, e; e_1, e_2)$ and *Removepoint*(e, r) by using *Insert_edge* and *Remove_edge*. However, explicit calls to *Remove_edge* cannot be made because the underlying graph for the subdivision may become disconnected during the process. Therefore, we will manipulate the data structure directly to simulate the effect of *Insert_edge* and *Remove_edge*. *Insertpoint*$(v, e; e_1, e_2)$, where $e = \{u, w\}$, is implemented by deleting e followed by inserting e_1 and e_2 in $\Im$. We then locate the two copies of e in the corresponding concatenable queues, split each into two leaves for e_1 and e_2, and rebalance if necessary. *Removepoint*$(v; e)$, where $e = \{u, w\}$, is implemented by deleting $\{u, v\}$ and $\{v, w\}$ from $\Im$, followed by inserting $\{u, w\}$ in $\Im$. We then delete the two copies of $\{u, v\}$ in the corresponding concatenable queues and rename the two copies of $\{v, w\}$ as $\{u, w\}$. Clearly, *Insert_point* and *Remove_point* run in $O(\log n)$ time.

Movepoint$(v; x, y)$ is implemented by deleting from $\Im$ the two old edges incident to v and inserting the two new ones in $\Im$. Since the resulting subdivision is topologically unchanged, no concatenable queue needs to be updated. Thus, *Movepoint*$(v; x, y)$ runs in $O(\log n)$ time.

THEOREM 2. *There is an $O(n)$-space data structure for dynamically maintain a connected subdivision. The data structure supports a Locate(p) query in $O(\log^2 n)$ time and the update operations Insert_edge$(u, w, r; r_1, r_2)$, Remove_edge$(e; r)$, Insertpoint$(v, e; e_1, e_2)$, Removepoint$(v; e)$, and Movepoint$(v; x, y)$ in $O(\log n)$ time. All time bounds are worst-case.*

COROLLARY 2. *An arbitrary chain of length k can be inserted (respectively, deleted) in $O(k \log(n + k))$ (respectively, $O(k \log n)$) worst-case time.*

Proof. Perform an insertion/deletion of each edge of the chain in $\Im$ successively. □

Hence, inserting/deleting a chain of length $o(\log n)$ can be done in $o(\log^2 n)$ time. This is an improvement upon the result presented in [19] when k is $o(\log n)$. Moreover, in [19] the chain is required to be monotone, whereas the chain can be arbitrary in our case.

4.2. Scheme II: Efficient insertion/deletion of monotone chains. We first assume that the input is specified as pointers to vertices and/or edges in the structure AL. If necessary, this assumption can be removed, as shown later, with a slight decrease in performance.

A chain is *monotone* if the y-coordinate is nondecreasing or nonincreasing when we walk along the chain from one end vertex to the other. Define a *maximal monotone chain* in the current subdivision to be a monotone chain that is not a proper subchain of a longer monotone chain.

A first attempt is to store maximal monotone chains instead of individual edges in the PSTs of $\Im$. It then becomes possible to insert/delete a monotone chain as a single entity instead of as individual edges. Each maximal monotone chain, γ, stored in the PSTs is represented as a concatenable queue that is sorted by the y-coordinates of the vertices of the chain. For each node $v \in B$, we extend the definition of $S(v)$ such that $S(v)$ is the set of maximal monotone chains that intersect $Y(v)$ but not $Y(u)$ for all proper ancestors u of v. Each maximal monotone chain, $\gamma \in S(v)$, intersects $Y(v)$ at a point $(x_\gamma(v), y(v))$. Let the endpoints of γ be $(x_l(\gamma), y_l(\gamma))$ and $(x_r(\gamma), y_r(\gamma))$, where $y_l(\gamma) \geq y_r(\gamma)$. Then we store the pair $(x_\gamma(v), y_l(\gamma) - y(v))$ in $L_1(v)$ (or $L_2(v)$) and the pair $(x_\gamma(v), y(v) - y_r(\gamma))$ in $R_1(v)$ (or $R_2(v)$). The rest of the organization is similar to that of Scheme I.

In both *Find* and *Fixup*2 we were able to check the position of a point relative to an edge and compute the intersection of an edge with a horizontal line in $O(1)$ time, but now these have to be done by binary searching on a maximal monotone chain and may take up to $O(\log n)$ time. Therefore, the modified *Fixup*2 takes $O(\log n)$ time, and the modified *Find* takes $O(\log^2 n)$ time. Hence, a point location query now takes $O(\log^3 n)$ time and a single insertion/deletion operation in $\Im$ takes $O(\log^2 n)$ time. (To insert a chain γ into $\Im$, we locate the node v such that $\gamma \in S(v)$ by spending $O(1)$ time per node visited. We then compute $x_\gamma(v)$ in $O(\log n)$ time by binary search on γ. However, the insertion time for $\Im$ is still dominated by *Fixup*2.)

Consider inserting a monotone chain, $\gamma = u_1 \cdots u_2$, of length k. It is possible that γ extends a maximal monotone chain, or γ joins two maximal monotone chains to become a longer maximal monotone chain, or γ is a maximal monotone chain by itself. Furthermore, γ may split another maximal monotone chain that contains an end vertex

of γ. Therefore, we may have to do a constant number of splittings/mergings of concatenable queues for the maximal monotone chains involved. Also, we have to delete the corresponding old maximal monotone chains from $\Im$ and insert the new ones into $\Im$. Clearly, it takes $O(\log^2 n)$ time to do all this. We then update the concatenable queues for the regions in $O(\log n + k)$ time by a simple generalization of the strategy discussed in §4.1. Hence, the total time for inserting a monotone chain is $O(\log^2 n + k)$. Deletion is essentially the reverse of insertion, and can also be implemented in $O(\log^2 n + k)$ time.

Thus, we have extended Scheme I so that insertion/deletion of a k-edge monotone chain takes $O(\log^2 n + k)$ time, while a point location query takes $O(\log^3 n)$ time. The $\log n$ blow-up comes from the fact that we may have to do a binary search on a maximal monotone chain of length $\Theta(n)$. This suggests that it may be possible to improve both time bounds if we divide a maximal monotone chain into smaller pieces and store these. However, the division should not be so fine that we end up spending more than $O(k)$ time to insert/delete a k-edge monotone chain.

Formally, we introduce a parameter b and choose its current value to be some positive integer n_0. Suppose that $n_0/4 \leq n \leq 3n_0$, where n is the number of vertices currently in the subdivision. Let $h(n)$ be the function $\max\{\log n \log\log n, 1\}$. Each maximal monotone chain of length at least $h(b)$ is divided into subchains whose length is in the range $[h(b), 2h(b)]$. Each such subchain is called a *monotone subchain*. Every maximal monotone chain of length less than $h(b)$ is also defined to be a *monotone subchain*. We call this an $h(b)$-*split* of the subdivision. Then, analogous to the previous approach, we store monotone subchains in $\Im$ instead of individual edges. Each monotone subchain, σ, stored in the PSTs is represented as a concatenable queue that is sorted by the y-coordinates of the vertices of the subchain.

Computing $x_\sigma(v)$ for a monotone subchain σ, while inserting σ into PSTs at v, now takes $O(\log\log n_0) = O(\log\log n)$ time, and so a single insertion operation into $\Im$ takes an extra $O(\log\log n)$ time. The time complexities of a *Find* and *Fixup*2 will now increase to $O(\log n \log\log n_0) = O(h(n))$ and $O(\log\log n_0) = O(\log\log n)$, respectively. Hence, a point location query takes $O(h(n)\log n)$ time and a single insertion/deletion operation in $\Im$ takes $O(h(n))$ time.

We first describe an updating method that is not quite correct but provides the basic intuition. Subsequently, we show how to fix the flaw. Also, the updating of the concatenable queues for the regions in the subdivision is not very difficult and can be done in $O(\log n + k)$ time, so we shall omit the details. Consider inserting a monotone chain of length k. If $k < h(b)$, we simply consider it as a single monotone subchain. If $k \geq h(b)$, we chop off at one end a monotone subchain of length $h(b)$. In either case, let σ be the monotone subchain obtained. We insert σ into the data structure according to the following cases. Note that we have to maintain an $h(b)$-split of the subdivision. To this end, we use the procedure *Resplit*.

1. σ joins only one monotone subchain σ_1 to form a longer monotone chain $\sigma_1\sigma$ (or $\sigma\sigma_1$). If $|\sigma_1| < h(b)$ or $|\sigma| < h(b)$, then delete σ_1 from $\Im$ and call *Resplit*$(\sigma_1\sigma)$ (or *Resplit*$(\sigma\sigma_1)$). Otherwise, insert σ into $\Im$.

2. σ joins together two monotone subchains, σ_1 and σ_2, to form a longer monotone chain $\sigma_1\sigma\sigma_2$. If $|\sigma_1| < h(b)$ and $|\sigma_2| < h(b)$, then delete σ_1 and σ_2 and call *Resplit*$(\sigma_1\sigma\sigma_2)$. Else, if $|\sigma_i| < h(b), i = 1$ or 2, then delete σ_i from $\Im$ and call *Resplit*$(\sigma_1\sigma)$ if $i = 1$ or *Resplit*$(\sigma\sigma_2)$ if $i = 2$. Otherwise, if $|\sigma| < h(b)$, then delete σ from $\Im$ and call *Resplit*(σ). If none applies, then insert σ into $\Im$.

3. σ is a maximal monotone chain by itself. Insert σ into $\Im$.

Furthermore, if an end vertex u_1 of σ lies on a monotone subchain $\sigma' \neq \sigma$, then

we delete σ' from $\Im$, split σ' into two pieces, σ'_1 and σ'_2, at u, and call *Resplit*(σ'_1) and *Resplit*(σ'_2). If the two end vertices, u_1 and u_2, of σ lie on the same monotone subchain $\sigma' \neq \sigma$, then we delete σ' from $\Im$, split σ' at u_1 and u_2 into three pieces σ'_1, σ'_2, and σ'_3 (σ'_2 has u_1 and u_2 as end vertices), and call *Resplit*(σ'_1) and *Resplit*(σ'_3). We repeat the operation of chopping the monotone chain and processing of monotone subchains obtained until a monotone chain of length less than $2h(b)$ is obtained. We then take this chain as a monotone subchain and process it as in the above.

Procedure *Resplit* is as follows. It is assumed that the length of any chain γ passed to *Resplit* is at most $4h(b)$, but not in the range $[h(b), 2h(b)]$. If γ is a maximal monotone chain, then we just insert it into $\Im$. Otherwise, we merge γ with some adjacent monotone subchains, if necessary, and then resplit the resulting monotone chain into two monotone subchains of the correct lengths.

```
Procedure Resplit(γ)
    done:= false
    While not done do
        done:= true
        if |γ| < h(b) then
            if there is a monotone subchain σ attached to γ such that
                σγ (or γσ) is a monotone chain then
                delete σ from ℑ
                γ := σγ (or γσ)
                done:= false
            else
                insert γ into ℑ
            end if
        else
            if |γ| > 2h(b) then
                split γ into two equal-sized pieces γ′ and γ″
                insert γ′ and γ″ into ℑ
            else
                insert γ into ℑ
            end if
        end if
    end while
end Resplit
```

If $n+k$ does not exceed $3n_0$, the value of b can be fixed to be n_0 throughout. Since we have an $h(b)$-split before the insertion, a chain γ passed into *Resplit* will not go through more than two iterations of the while loop in *Resplit*. Therefore, *Resplit* runs in $O(h(n))$ time. Thus, we insert $O(k/h(n_0)) = O(k/h(n))$ monotone subchains, and each takes $O(h(n+k)) = h(n)$ time. This gives a worst-case time bound of $O(h(n)+k)$ for insertion.

Let $\sigma = u_1 \cdots u_2$ be a monotone chain to be deleted. In general, σ is of the form $u_1 \cdots v_1 \sigma_1 \cdots \sigma_m v_2 \cdots u_2$, where σ_i, $1 \leq i \leq m$, is a monotone subchain in the current $h(b)$-split. Each σ_i is deleted from $\Im$. If $|\sigma_i| = \Theta(h(n))$ for all i, then we would be deleting $O(k/h(n_0)) = O(k/h(n))$ monotone subchains, and each takes $O(h(n))$ time. There are two cases for deleting $u_1 \cdots v_1$:

1. If $u_1 \cdots v_1$ is a monotone subchain by itself, then delete $u_1 \cdots v_1$ from $\Im$. If the degree of u_1 does not become two, then we are done. Else, if the two monotone sub-

chains σ' and σ'' attached to u_1 cannot be joined together to form a monotone chain, then again we do nothing further. Otherwise, if either $|\sigma'|$ or $|\sigma''|$ is less than $h(b)$, then we delete σ' and σ'' from $\Im$ and call *Resplit*$(\sigma'\sigma'')$;

2. If $u_1 \cdots v_1$ is the suffix of a longer monotone subchain σ', we delete σ' from $\Im$. Then we split σ' at u_1 into two pieces, σ'' and $u_1 \cdots v_1$, and call *Resplit*(σ'') if $|\sigma''| < h(b)$.

Again the call to *Resplit* in either of the two cases involves no more than two iterations of the while loop, and thus both cases take $O(h(n))$ time. We handle $v_2 \cdots u_2$ in an analogous fashion. This gives a worst-case time bound of $O(h(n) + k)$ for deletion.

Notice that an insertion/deletion of a monotone subchain may cause extra updates in $\Im$ because of the calls to *Resplit*. We call the insertion/deletion of a monotone subchain with the extra updates in $\Im$ that follow a *complete update*. (We shall also use the more refined terms, *complete insertion* and *complete deletion*, which have obvious meanings.) Moreover, the extra updates induced do not modify the underlying subdivision, and hence n, the number of vertices in the current subdivision, is also not changed by these extra updates.

The updating strategy in the above has a major flaw as pointed out below. After a sequence of complete insertions, n may have increased to the extent that the length of some existing monotone subchains may become much less than $h(n)$. Therefore, a complete deletion operation may then involve deleting more than $O(k/h(n))$ monotone subchains, and thus the $O(h(n) + k)$ bound is no longer valid. On the other hand, after a sequence of complete deletions, n may have decreased to the extent that the length of some existing monotone chains may become much more than $O(h(n))$. Therefore, it may then take more than $O(\log\log n)$ time for a call to *Fixup*2 and more than $O(h(n))$ time for a call to *Find*. This will cause both the future update time and query time to deteriorate. As a remedy, we must adjust the parameter b whenever necessary. Changing the value of b will then force us to redo the decomposition of some maximal monotone chains currently stored. We introduce two additional structures that will facilitate the redecomposition. We maintain a doubly linked list, *LONG*, of those monotone subchains whose lengths are in the range $[h(b), 2h(b)]$. Therefore, the size of *LONG* is at most $n/h(b)$ at any time. The remaining monotone subchains, whose lengths are less than $h(b)$, are stored in a max heap, *SHORT*, according to their lengths. All the monotone subchains in *SHORT* are maximal monotone chains. Monotone subchains should be inserted/deleted in *LONG* or *SHORT* as they are inserted/deleted in $\Im$ (insertions of monotone subchains into *LONG* take place at the front of *LONG*). We associate pointers from subchains in $\Im$ to their copies in *LONG* or *SHORT* to avoid searching. Thus updates in *LONG* and *SHORT* take $O(1)$ time and $O(\log n)$ time, respectively.

We pick a new value for b and start the redecomposition as soon as n exceeds $2b$ or drops below $b/2$. Note that throughout the redecomposition, n must not exceed $2b'$ or drop below $b'/2$, where b' is the new b, to avoid starting a new redecomposition before the previous one ends.

Let the initial value of b be n_0. If n exceeds $2b$ at some point of time, we assign b the value $2n_0$. Note that n is at most $2n_0 + 2h(n_0)$ at this point. The lengths of some monotone subchains in *LONG* may become less than $h(b)$, but their lengths can still be generously bounded from below by $h(b)/4$. We walk through *LONG*, and for each such monotone subchain σ, we delete σ from *LONG*, call *Resplit*(σ), and add the two new monotone subchains generated to the front of *LONG*. Since every monotone subchain in the subdivision has length at least $h(b)/4$, a call to *Resplit*(σ) will not involve more than four iterations of the while loop. Therefore, each call takes $O(\log n \log\log b)$ time because the skeleton B of $\Im$ has height $O(\log n)$, and each monotone subchain stored in

the PSTs of $\Im$ has length at most $2h(b)$. We spread the processing of *LONG* over the next $n_0/2h(b) - 1$ complete updates. Since *LONG* contains at most $n/h(n_0) \leq 12n_0/h(b)$ elements, the number of elements in *LONG* checked after each such complete update is bounded by a constant. After we have scanned the entire list *LONG*, the value of n must be in the range $[n_0, 3n_0] = [b/2, 3b/2]$. As a result, during this period the length of each monotone subchain processed is $\Theta(h(n))$, each complete update takes $O(h(n))$ time, and a new $h(b)$-split is obtained at the end.

On the other hand, if the initial value of b is n_0 and n becomes less than $b/2$ at some point of time, we assign b the value $n_0/2$. Note that n is at least $n_0/2 - 2h(n_0)$ at this point. The lengths of some monotone subchains in *LONG* may become greater than $2h(b)$, but they can still be generously bounded from above by $8h(b)$. Then, as before, we walk through *LONG* and split all such monotone subchains. Moreover, some monotone subchains in *SHORT* may become long enough to be put in *LONG*. We move them to *LONG* by repeatedly deleting the monotone subchain stored at the root of *SHORT* and adding it to front of *LONG* until the length of the monotone subchain stored at the root is less than $h(b)$. This processing is spread over the next $n_0/8h(b) - 2$ complete updates. Since the total number of monotone subchains we have looked at in *LONG* and *SHORT* must be at most $n/h(b) \leq 3n_0/h(b)$, the number of monotone subchains that need to be checked after each complete_update is bounded by a constant. After we have scanned the entire list *LONG* and extracted all the eligible monotone subchains in *SHORT*, the value of n must be in the range $[n_0/4, 3n_0/4] = [b/2, 3b/2]$. Consequently, each complete update during this period takes $O(h(n))$ time, and a new $h(b)$-split is obtained at the end.

In summary, we have proved that a k-edge monotone chain can be inserted/deleted in $O(\log n \log\log n + k)$ time. Since a single edge can be treated as a monotone chain of length one and *Insertpoint*, *Removepoint*, and *Movepoint* involve simulations of a constant number of insertions and deletions of edges, they take $O(\log n \log\log n)$ time.

THEOREM 3. *There is an $O(n)$-space structure for dynamically maintaining a connected planar subdivision. If the input is specified as pointers to vertices and edges in the underlying graph, then the structure yields a query time of $O(\log^2 n \log\log n)$ for answering a point location query, an update time of $O(\log n \log\log n)$ for inserting/deleting/moving of a degree-2 vertex, and an update time of $O(\log n \log\log n+k)$ for inserting/deleting a monotone chain of length k. All bounds are worst-case.*

If the chain is not monotone, then we split it into monotone pieces and invoke the above procedures for inserting/deleting each of those monotone pieces.

COROLLARY 3. *If the input is specified as pointers to vertices and edges in the underlying graph, then an arbitrary chain can be inserted (respectively, deleted) in $O(k \log(n + k) \log\log(n + k))$ (respectively, $O(k \log n \log\log n)$) worst-case time.*

If only coordinates are given in the input, then at first sight it appears that we could access the corresponding vertices and edges in AL by using the dictionaries D_V and D_E. However, we cannot keep all the vertices and edges of the subdivision in D_V and D_E since otherwise we need to spend $O(k \log n)$ time to update D_V and D_E after an insertion/deletion of a k-edge monotone chain. The lack of efficient access to the data structure makes the updating of the data structure problematic. For instance, consider inserting a monotone chain $\gamma = u_1 \cdots u_2$. Suppose that u_1 is a nonextreme vertex of a monotone subchain σ. We then need to split the concatenable queue for σ. However, we do not know where to do the splitting since we do not have access to the two edges in σ that are incident to u_1. This problem can be solved by performing a point location query for u_1 using our data structure before inserting the input monotone chain γ. This query will locate in $\Im$ one of the two edges of σ that are incident to u_1. By following the

appropriate pointers associated with that edge in $\Im$, we can thus locate the position to split the concatenable queue for σ (and also the concatenable queue for the region containing γ if necessary). Because of the point location query, the insertion time becomes $O(\log^2 n \log\log n + k)$. Deletion of a monotone chain can be handled similarly and takes $O(\log^2 n \log\log n + k)$ time.

THEOREM 4. *There is an $O(n)$-space structure for dynamically maintaining a connected planar subdivision. If only coordinates are specified in the input, then the structure yields a query time of $O(\log^2 n \log\log n)$ for a point location query, an update time of $O(\log^2 n \log\log n)$ for inserting/deleting/moving of a degree-2 vertex, and an update time of $O(\log^2 n \log\log n + k)$ for inserting/deleting a monotone chain of length k. All bounds are worst-case.*

COROLLARY 4. *If only coordinates are specified in the input, then an arbitrary chain can be inserted (respectively, deleted) in $O(k \log^2(n+k) \log\log(n+k))$ (respectively, $O(k \log^2 n \log\log n)$) worst-case time.*

4.3. Generalizing to algebraic segments. Scheme I can be generalized to handle planar subdivisions that consist of algebraic segments of bounded degree d. Algebraic segments of bounded degree are also considered in a different context by Mulmuley [11]. We follow the same purely algebraic approach presented in [11]. A monotone algebraic segment, s, satisfying the equation $f(x, y) = 0$ can be specified uniquely by its two extreme points (x_s, y_s) and (x'_s, y'_s), where $y_s \geq y'_s$, and the orientation of the tangent to s at (x_s, y_s). If s is not monotone, then we divide s into $O(d^2)$ monotone pieces by cutting at the local extrema on the segment where the tangents become horizontal. These points satisfy the equations $f(x, y) = 0$ and $\frac{\partial f}{\partial x} = 0$, and can be computed as described in [11]. We then find out, using the strategy given in [11], how these monotone pieces are topologically connected together. Hence, we can treat the subdivision as a collection of monotone algebraic segments and store these segments in $\Im$. An algebraic segment can thus be inserted/deleted by dividing it into monotone pieces as above and then inserting/deleting each monotone piece in $\Im$. Note that *Find* and *Fixup*2 will involve computing the intersection between a monotone algebraic segment, s, and a horizontal line. Except for the operation of moving a degree-2 vertex, whose semantics are not well defined in this case, all the other operations supported by Scheme I can be handled analogously. Let $T_1(d)$ be the time to compute the intersection between a monotone segment and a horizontal line, and let $T_2(d)$ be the time to divide an algebraic segment into monotone pieces. Then the above discussion implies that the variation uses $O(d^2 n)$ space, takes $O(T_1(d) \cdot \log^2(d^2 n))$ time for answering a point location query, $O(T_2(d) + T_1(d) \cdot d^2 \log(d^2 n))$ time for inserting/deleting a segment, and $O(T_2(d) + T_1(d) \cdot d^2 \log(d^2 n))$ time for inserting/deleting a point on a segment. Furthermore, it is in fact possible to maintain subdivisions consisting of any type of curve segments as long as there are efficient ways to divide those curve segments into a few monotone pieces and to compute the intersection between a monotone segment and a horizontal line.

5. Persistent planar point location. In this section, we show how to extend Scheme I to support a point location query with respect to the subdivision at any time t' in the past, while updates are allowed in the present. In other words, we seek to develop a *persistent* representation of connected subdivisions. The problem of maintaining a subdivision persistently was first proposed by Sarnak and Tarjan [22]. The extension is based on applying the techniques of path copying [6], [12], [13], [21], [23] and limited node copying [22]. Let n be the number of vertices in the subdivision, and let m be the number of

updates that have occurred so far. We assume that we start with an empty subdivision, and thus m is greater than or equal to n at all times. To avoid ambiguity, we use t to denote the current time, subscript t' to denote the latest version of an object at time t' in the past, and accent "ˆ" to denote the latest version of an object.

$\Im$ is organized as in the amortized version of Scheme I (§3.1.1), namely, each node v of B has two PSTs $L(v)$ and $R(v)$ for storing the edges in $S(v)$. To insert an edge e, we first insert into B as before the two leaves corresponding to the two end vertices of e. We then insert e into $L(v)$ and $R(v)$, where $e \in S(v)$. Since we are to support querying in the past, the insertion/deletion must be done in such a way that we do not lose the old $L(v)$ and $R(v)$. Take $\hat{L}(v)$ as an example. We binary search down $\hat{L}(v)$ to insert the leaf, w, for $(x_l(e), y_l(e))$ and then duplicate the path from w to the root to produce $L_t(v)$. The root of $L_t(v)$ is then stamped with the current time t, and a pointer to it is inserted into a dictionary, $D(v)$, sorted in decreasing order of time stamps. Next, we have to maintain the heap order on the priority field. Let a' be the node from w to the root of $L_t(v)$ such that $a'.y < w.y$ but $parent(a').y \geq w.y$. Therefore, a' should be filled with the content of w', which in turn forces us to fill an appropriate child of a' with the content of a' to make room for w. This down-shifting step repeats until we reach a leaf. In general, the down-shifting may first follow the path from a' to w and then deviate to continue along a totally separate path. Since the path traversed after the deviation belongs to the previous version of $L(v)$, we must copy the nodes visited to preserve the past information. It should be clear that the total time taken is $O(\log m)$. (It takes $O(\log m)$ time to insert the two leaves into B, $O(\log \hat{n})$ time to do path copying, and updating in $\hat{L}(v)$ and $\hat{R}(v)$, as well as $O(\log m)$ time to insert a pointer to the root of $L_t(v)$ into $D(v)$.) The space needed is $O(\log \hat{n})$ worst-case due to copying at most two paths in each of $\hat{L}(v)$ and $\hat{R}(v)$. B may become unbalanced after this insertion, and, therefore, we may need to perform rotations at certain nodes visited. We need to reconstruct the structures $L_{t'}(v)$, $R_{t'}(v)$ and $D(v)$ for all t' for each node v that actively participates in a rotation. Moreover, the reconstruction of a PST cannot proceed in a bottom-up fashion as before since, otherwise, it becomes impossible to distinguish among the different versions of $L(v)$ and $R(v)$ at different times. Therefore, we implement the reconstruction by repeatedly inserting the edges in the order of increasing time stamps. As proved in [25], this insertion strategy takes $O(\log^2 m)$ amortized time. The space needed by the reconstruction can similarly be proved to be $O(1)$ amortized. To delete an edge e, $e \in \hat{S}(v)$, we first duplicate the path from the root to the leaf for $(x_l(e), y_l(e))$ (respectively, $(x_r(e), y_r(e))$) to generate $L_t(v)$ (respectively, $R_t(v)$), and then we actually delete $(x_l(e), y_l(e))$ (respectively, $(x_r(e), y_r(e))$) from $L_t(v)$ (respectively, $R_t(v)$). B is not modified at all. Clearly, the deletion process takes $O(\log \hat{n})$ worst-case space and $O(\log \hat{n} + \log m) = O(\log m)$ time.

Querying $\Im$ at time t' proceeds as follows. We traverse a root to leaf path in B as before. At each node v visited, we query $D(v)$ in $O(\log m)$ time to access the root of $L_{t'}(v)$ (or $R_{t'}(v)$). We then query $L_{t'}(v)$ (or $R_{t'}(v)$) in $O(\log n_{t'})$ time, which is dominated by $O(\log m)$. Thus, the overall query time is $O(\log^2 m)$.

We can reduce this bound to $O(\log n_{t'} \cdot \log m)$ by *layering* the dictionaries. This would have the effect of reducing the dictionary access time from $O(\log m)$ to $O(1)$ for the nonroot nodes on the path traversed. Specifically, at each node v that is not the root of B, $D(v)$ is organized as a linked list of pointers in sorted order of decreasing time stamps. Moreover, we keep two *down* pointers at each entry time stamped t' in $D(v)$ to the two entries time stamped t_1 and t_2 in $D(left(v))$ and $D(right(v))$, respectively, where $t_1 = \max\{x : x \leq t'$, and x is a time stamp in $D(left(v))\}$ and $t_2 = \max\{x : x \leq$

t', and x is a time stamp in $D(\mathit{right}(v))\}$. At the root r of B, $D(r)$ is organized as an balanced search tree. However, since insertion/deletion of an edge e takes place at a single node v of B, the time stamps of all the entries in $D(\mathit{parent}(v))$ are less than t, and thus there is no *down* pointer from $D(\mathit{parent}(v))$ to the new entry added to $D(v)$. If more updates occur at v, then the possibility of reducing the access time for $D(v)$ to $O(1)$ will be ruled out. We can add dummy entries time stamped t to the dictionary of each ancestor of v after every update at v, but this has the undesirable effect of increasing the space bound per update from $O(\log \hat{n})$ to $O(\log m)$. Instead, we apply limited node copying [22] to obtain an $O(1)$ amortized space bound per update for layering the dictionaries. Specifically, at each node v, whenever both the first and second entries of $D(v)$ (i.e., the entries with the largest and second largest time stamps) are not pointed to by *down* pointers from $D(\mathit{parent}(v))$, we add a dummy entry to the front of $D(\mathit{parent}(v))$, assign it a time stamp equal to that of the first entry of $D(v)$, and associate with it two *down* pointers to the first entries of $D(\mathit{left}(\mathit{parent}(v)))$ and $D(\mathit{right}(\mathit{parent}(v)))$. If necessary, we propagate the addition of a dummy entry to the parent of $\mathit{parent}(v)$, and so on. To analyze the amortized space needed, we follow the banking account paradigm [24]. We maintain the following invariant: At each node of B, the most recent dictionary entry that is not pointed by a *down* pointer has a token stored at it. Let an update occur at a node v of B. There may be a propagation of additions of dummy entries to dictionaries at some ancestors of v. Suppose that the propagation stops at a node u and adds a dummy entry *item* to $D(u)$. For each $u' \neq u$ on the path from v to u, adding a new dummy entry to $D(u')$ can be paid for with the token stored at that entry's predecessor (the token must exist since the predecessor cannot be pointed to by a *down* pointer). Moreover, no token needs to be stored at this new dummy entry because it must be pointed to by a *down* pointer. At u we store a new token at *item* to preserve the invariant. It follows that the amortized space cost for an update is $O(1)$. During querying, we may reach at a dummy entry in $D(v)$, therefore, we keep a *shift* pointer at each dummy entry of $D(v)$ to the nearest nondummy entry with a smaller time stamp so that we can access the roots of the desired version of $L(v)$ and $R(v)$ in $O(1)$ time.

The final procedure for querying $\Im$ at time t' works as follows. We traverse a root to leaf path in B as before. We query $D(r)$ to retrieve an entry *item*. Let v be r initially. If *item* is nondummy, then we can access the roots of $L_{t'}(v)$ and $R_{t'}(v)$. Otherwise, we follow the *shift* pointer to locate the nearest nondummy entry with a smaller time stamp. We then query either $L_{t'}(v)$ or $R_{t'}(v)$ as appropriate. Next, we follow the *down* pointer stored at *item* to get an entry *candidate* in the dictionary at the next node v' on the path. It must be the case that either *candidate* or the entry just in front of *candidate* has the largest time stamp smaller than t'. We set *item* to the appropriate one of these, set v to v', and then repeat the above. Hence, we spend $O(\log m)$ time in querying $D(r)$ and $O(1)$ time, plus the time for querying either $L_{t'}(\cdot)$ or $R_{t'}(\cdot)$ at each of the $O(\log m)$ nodes on the path. The query time thus sums up to $O(\log n_{t'} \cdot \log m)$.

Insertion/deletion of an edge e at a node v of $\Im$ then involves the additional steps of adding an entry time stamped t to the front of $D(v)$, setting its *down* pointer equal to that of the second entry, and adding dummy entries time stamped t to the dictionaries at its ancestors whenever necessary. These additional steps clearly take $O(\log m)$ worst-case time, and the amortized space needed is $O(1)$ as mentioned above. Hence, we have improved the query time from $O(\log^2 m)$ to $O(\log n_t \cdot \log m)$ without affecting the time and space bounds for insertion and deletion in $\Im$.

Note that the $O(\log^2 m)$ amortized time bound for insertion can be turned into worst-case by spreading the reconstruction over a sequence of future updates. Both

the delete and query procedures have to be changed slightly, but the respective time and space bounds will not be affected. The details resemble the strategy presented in §3.3, and we omit the discussion here.

In summary, we have obtained a persistent structure for the dynamic visibility problem, yielding a query time of $O(\log n_t \cdot \log m)$, an insertion time of $O(\log^2 m)$, a deletion time of $O(\log m)$, and using $O(\log \hat{n})$ worst-case plus $O(1)$ amortized space per update. All time complexities are worst-case.

The concatenable queues representing the regions can similarly be made persistent by applying path copying. The space needed per update is thus $O(\log \hat{n})$ worst-case. A node in the concatenable queue may now have more than one parent (with different time stamps), and we store the parent pointers of a node in a dictionary sorted in decreasing order of time stamps. Because of path copying, we may need to insert new parent pointers into as many as $\Theta(\log \hat{n})$ such dictionaries during a join or split operation. If the dictionary of parent pointers at each node is organized as a balanced search tree, then it takes a total of $O(\log \hat{n} \cdot \log m)$ time, which dominates the run time of a join or split operation. Note, however, that we know the actual position of each new parent pointer in the corresponding dictionary because its time stamp is larger than any existing one. Therefore, we can organize the dictionary using the data structure developed recently in [8], which supports member and neighbor queries in $O(\log m)$ worst-case time and allows for $O(1)$ worst-case update time once the position of element to be inserted or deleted is known. Hence, the time for a join or split operation can be improved to $O(\log \hat{n})$. To report the name of a region to the left of an edge e at time t', we simply traverse a leaf to root path via the proper parent pointers. Retrieving each such pointer from the dictionary at each node takes $O(\log m)$ time, and hence a total of $O(\log n_t \cdot \log m)$ time.

Combining the persistent structure for the dynamic visibility problem with the persistent concatenable queues, we obtain the following result.

THEOREM 5. *Starting with an empty subdivision, there is a persistent structure for maintaining connected subdivisions, supporting an $O(\log n_{t'} \cdot \log m)$-time query with respect to the subdivision at time t', an insertion time of $O(\log^2 m)$, and a deletion time of $O(\log m)$, and using $O(\log \hat{n})$ worst-case plus $O(1)$ amortized space per update, where m is the total number of updates that have occurred, $\hat{n}$ is the number of vertices in the subdivision at present, and $n_{t'}$ is the number of vertices in the subdivision at time t'. All time complexities are worst-case and updates are allowed only in the present.*

6. Conclusion and discussion. We have presented an efficient dynamic point location scheme for connected planar subdivisions. It extends and/or improves upon existing dynamic planar point location schemes. In particular, we show a remarkable update time of $O(\log n)$ for inserting/deleting an edge, while still being able to answer a point location query in $O(\log^2 n)$ time. We then show how to speed up the insertion/deletion time for a k-edge monotone chain at the expense of increasing the other time bounds slightly. Moreover, our result can be generalized to cope with connected subdivisions consisting of algebraic segments of bounded degree, and it can also be extended to support persistent planar point location. The main component of the above results is a new structure for a planar dynamic visibility problem for line segments that are nonintersecting, except possibly at endpoints. In fact, this structure can be used even when the subdivision is *nonconnected*. The reason that our technique is limited to connected subdivisions is the lack of an efficient structure to maintain a nonconnected subdivision and report the name of the region on one side of a given edge. Obtaining such a structure is an interesting research problem. Another open question is whether the query time can be reduced to $o(\log^2 n)$ with little or no increase in the update time.

REFERENCES

[1] R. Cole, *Searching and storing similar lists*, J. Algorithms, 7 (1986), pp. 202–220.

[2] D. Dobkin and R. Lipton, *Multidimensional searching problems*, SIAM J. Comput., 5 (1976), pp. 181–186.

[3] H. Edelsbrunner, L. Guibas, and J. Stolfi, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–340.

[4] O. Fries, K. Mehlhorn, and S. Naeher, *Dynamization of geometric data structures*, in Proc. of 1st ACM Symposium on Computational Geometry, Baltimore, MD, 1985, pp. 168–176.

[5] D. Kirkpatrick, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.

[6] T. Krijnen and L. Meertens, *Making B-trees work for B*, Tech. Report IW 219/83, The Mathematical Centre, Amsterdam, the Netherlands, 1983.

[7] D. T. Lee and F. Preparata, *Location of a point in a planar subdivision and its applications*, SIAM J. Comput., 6 (1977), pp. 594–606.

[8] C. Levcopoulos and and M. Overmars, *A balanced search tree with $O(1)$ worst-case update time*, Acta Inform., 26 (1988), pp. 269–277.

[9] E. McCreight, *Priority search trees*, SIAM J. Comput., 14 (1985), pp. 257–276.

[10] K. Mehlhorn, *Data Structures and Algorithms: Vol.* 3, *Multidimensional Searching and Computational Geometry*, Springer-Verlag, New York, 1984.

[11] K. Mulmuley, *A fast planar partition algorithm,* II, in Proc. 5th ACM Symposium on Computational Geometry, Saarbrucken, Germany, 1989, pp. 33–43.

[12] E. Myers, *AVL dags*, Tech. Report 82-9, Department of Computer Science, University of Arizona, Tucson, AZ, 1982.

[13] ———, *Efficient applicative data types*, in Conference Record Eleventh Annual ACM Symposium on Principles of Programming, 1984, Salt Lake City, UT, pp. 66–75.

[14] J. Nievergelt and E. Reingold, *Binary search trees of bounded balance*, SIAM J. Comput., 2 (1973), pp. 33–43.

[15] M. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, Springer-Verlag, New York, 1983.

[16] ———, *Range searching in a set of line segments*, in Proc. 1st ACM Symposium on Computational Geometry, Baltimore, MD, 1985, pp. 177–185.

[17] M. Overmars and J. van Leeuwen, *Worst-case optimal insertion and deletion methods for decomposable searching problems*, Inform. Process. Lett., 12 (1981), pp. 168–173.

[18] F. Preparata, *A new approach to planar point location*, SIAM J. Comput., 10 (1981), pp. 473–483.

[19] F. Preparata and R. Tamassia, *Fully dynamic point location in a montone subdivision*, SIAM J. Comput., 18 (1989), pp. 811–830.

[20] ———, *Dynamic planar point location with optimal query time*, Theoret. Comput. Sci., 74 (1990), pp. 95–114.

[21] T. Reps, T. Teitelbaum, and A. Demers, *Incremental context-dependent analysis for language-based editors*, ACM Trans. Prog. Lang. Syst., 5 (1983), pp. 449–477.

[22] N. Sarnak and R. Tarjan, *Planar point location using persistent search trees*, Comm. ACM, 29 (1986), pp. 669–679.

[23] G. Swart, *Efficient algorithms for computing geometric intersections*, Tech. Report #85-01-02, Department of Computer Science, University of Washington, Seattle, WA, 1985.

[24] R. Tarjan, *Amortized computational complexity*, SIAM J. Algebraic Discrete Meth., 6 (1985), pp. 306–318.

[25] D. Willard and G. Lueker, *Adding range restriction capability to dynamic data structures*, J. ACM, 32 (1985), pp. 597–617.

SIAM J. COMPUT.
Vol. 21, No. 5, p. 1000, October 1992

© 1992 Society for Industrial and Applied Mathematics
011

ADDENDUM: MINIMUM WEIGHTED COLORING OF TRIANGULATED GRAPHS, WITH APPLICATION TO MAXIMUM WEIGHT VERTEX PACKING AND CLIQUE FINDING IN ARBITRARY GRAPHS*

EGON BALAS† AND JUE XUE‡

This note is meant to answer the numerous queries we have received about the details of implementing Algorithm 3 in §5 of [1], in particular its version that produced the computational results of §6.

Algorithm 3 was implemented and tested in several versions. The version that proved most efficient, and on which the computational results of §6 are based, differs from the outline given in §5 in three respects:

1. In Step 0, an edge-maximal triangulated subgraph (EMTS) of G is generated, and its maximum weight clique is used to initialize the lower bound $w(K^*)$. The EMTS is generated with an $O(|E| \cdot \Delta)$ algorithm described in [3], which is an improvement over Algorithm 1 of [2]. (Here Δ is the maximum degree in G.)
2. The stated version of Step 2 is applied only when $w(K^*) \leq w(I_t)$ or $| V_t | < 30$; otherwise, Step 2 reduces to choosing an arbitrary ordering σ_t of V_t.
3. Step 3 uses a modified version of Algorithm 2 that starts with the sequence σ_t and generates a maximal induced subgraph $G(W_t)$ of G along with a weighted coloring of $G(W_t)$ equal to $w(K^*)$.

REFERENCES

[1] E. BALAS AND J. XUE, *Minimum weighted coloring of triangulated graphs, with application to maximum weight vertex packing and clique finding in arbitrary graphs*, SIAM J. Comput., 20(1991), pp. 209–221.
[2] E. BALAS, *A fast algorithm for finding an edge-maximal subgraph with a* TR*-formative coloring*, Discr. Appl. Math., 15(1986), pp. 123–134.
[3] J. XUE, *Fast Algorithms for Vertex Packing and Related Problems*, Ph.D. thesis, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, PA, 1991.

* Received by the editors July 28, 1992; accepted for publication July 28, 1992.
† Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213.
‡ Graduate School of Management, Clark University, Worcester, Massachusetts, 01610.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1001-1007, December 1992

© 1992 Society for Industrial and Applied Mathematics

001

REDUCING THE STEINER PROBLEM IN A NORMED SPACE*

D. Z. DU† AND F. K. HWANG‡

Abstract. Consider a set P of points in a normed space whose unit sphere is a d-dimensional symmetric polytope with $2d$ extreme points. This paper proves that there always exists a Steiner minimum tree whose Steiner points are located only at points whose coordinates appear in points of P. This generalizes a recent result of Snyder on d-dimensional rectilinear space, which itself extends Hanan's well-known and much quoted result on the rectilinear plane. Furthermore, the proof in this paper is much simpler than Snyder's proof, even considerably shorter than Hanan's proof. A consequence of this result is that the Steiner problem for P in such a space is reduced to a Steiner problem on graphs and is solvable by any existing Steiner graph algorithms. The paper also conjectures that such a reduction is impossible if the polytope has more than $2d$ extreme points and provides partial support for the conjecture.

Key words. Steiner tree, Minkowski space, normed space

AMS(MOS) subject classifications. 51M05, 52A21, 90B99

1. Introduction. Let P be a given set of points in a space S. A *Steiner minimum tree* (SMT) for P is a shortest network interconnecting P. A point in a tree interconnecting P with degree at least three and not in P is called a *Steiner point*. Some spaces have the nice properties that there exists an SMT whose edges all follow some specified directions, or whose Steiner points all belong to a specified set of points. For example, it is clear that we need only consider vertical and horizontal edges in two-dimensional rectilinear space. Hanan [4] proved that the Steiner problem is finite in that space by showing that there exists an SMT with no Steiner point having coordinates (x, y) such that either x or y does not appear as a coordinate for some point in P. Recently, Snyder [7] extended this result to general dimension rectilinear space. In this paper, with a much simpler argument, we further generalize these results to any normed space whose unit sphere is a d-dimensional symmetric polytope with $2d$ extreme points satisfying certain regularity conditions We also conjecture that the $2d$ extreme points is a necessary condition.

2. Direction of edges. Consider a normed space S_d whose distance function is specified by a unit sphere that is a d-dimensional polytope symmetric with respect to the original point. Assume that the polytope has $2m$ extreme points. Then the m lines connecting the original points with the $2m$ extreme points are called *diagonals*, and their directions are called *diagonal directions*.

THEOREM 1. *Let P be a set of points in S_d, $d \geqq 2$. Then there exists an* SMT *using only edges in diagonal directions.*

Proof. Since any polytope is the convex hull of its extreme points, any point x on the boundary of the unit sphere can be represented as a linear function of extreme points $x_1, \cdots, x_k$:

$$\vec{x} = a_1\vec{x}_1 + \cdots + a_k\vec{x}_k,$$

where $a_i \geqq 0$ and $a_1 + \cdots + a_k = 1$. For any edge $[A, B]$, let x be a boundary point such

* Received by the editors April 4, 1991; accepted for publication (in revised form) September 11, 1991.

† Department of Computer Science, Princeton University, Princeton, New Jersey 08854 and the Institute of Applied Mathematics, Academia Sinica, Beijing, China.

‡ AT&T Bell Laboratories, Murray Hill, New Jersey 07974.

that $\vec{x}$ is parallel to $[A, B]$. Then

$$\overrightarrow{AB} = |AB| \cdot \vec{x} = |AB| a_1 \vec{x}_1 + \cdots + |AB| a_k \vec{x}_k.$$

Therefore, we find a path from A to B consisting of k segments, where the ith segment is parallel to $\vec{x}_i$ and has length $a_i|AB|$. Since $\sum_{i=1}^{k} a_i = 1$, the path length is $|AB|$.

3. Reducing S_d to a grid. The approach used here is similar to the one used by Snyder [7] for d-dimensional rectilinear space, but much simpler and more general. Consider a given set P of points in S_d whose unit sphere has $2m$ extreme points, $m \geqq d$, such that any d diagonal directions are linearly independent. Call a hyperplane *diagonal* if it is parallel to the span of $d-1$ diagonal directions. Let $G_1(P)$ be the grid formed by the $\binom{m}{d-1}$ diagonal hyperplanes through each P-point. Define a *grid point* to be a point where d distinct diagonal hyperplanes intersect.

THEOREM 2. *If $m = d$, then there exists an* SMT *for P whose Steiner points are all grid points of $G_1(P)$.*

Proof. Call a coordinate *listed* if it appears in a P-point. Then a grid point has only listed coordinates. By Theorem 1 there exists an SMT T using only edges in diagonal directions. Let U be the set of unlisted coordinates occurring in Steiner points of T. Suppose that U is nonempty. We show that we can always reduce the cardinality of U by one. This completes the proof.

Let s be a Steiner point with an unlisted coordinate x_i in the ith diagonal direction. Consider the diagonal hyperplane parallel to the span of the other $d-1$ diagonal directions at s. Clearly, there exists no P-point on this hyperplane or x_i would be listed. Since the d diagonal directions are linearly independent, we can move this hyperplane with all edges on it along the ith diagonal direction (either side) until it hits a P-point. Firstly, we note that such a move is permissible since no P-point has been moved. Secondly, the number of segments of the ith diagonal direction touching the hyperplane on the two sides must be equal or we could move it towards one side to reduce the total length; a contradiction to the assumption that T is an SMT. Thus moving the hyperplane does not change the length of T. Finally, when the hyperplane hits a P-point, the new x_i is listed, but no new unlisted coordinate has been created.

The following corollary was first proved by Snyder [7] ($d = 2$ case by Hanan [4]).

COROLLARY. *For P a set of points in the d-dimensional rectilinear space, there exists an* SMT *whose Steiner points are all grid points of $G_1(P)$.*

Theorem 2 reduces the SMT problem to a finite problem and opens the door for using SMT graph algorithms to attack the SMT problem in S_d.

We conjecture that $m = d$ is also a necessary condition for Theorem 2.

CONJECTURE 1. *If $m > d$, then there exists a set P in S_d such that every* SMT *for P contains a Steiner point that is not a grid point of $G_1(P)$.*

As partial support to Conjecture 1, we show that for every $d \geqq 2$ there exists a space S_d with $m > d$ and a set P in S_d such that every SMT for P contains a Steiner point that is not a grid point of $G_1(P)$.

A *splitting tree* T_n for n points in the Euclidean plane is defined [5] as

(i) T_3 is an SMT for three corners of an equilateral triangle;

(ii) T_n, $n > 3$, can be obtained from T_{n-1} by splitting an endpoint w of T_{n-1} into two new edges $[u, w]$ and $[v, w]$, where u and v are new endpoints of T_n, such that

(a) The three edges meet at 120° at the splitting point;

(b) The lengths of these two new edges are equal and less than $\lambda_{n-1}/4$ with

$$\lambda_{n-1} = \min_{R_{n-1}} \{|R_{n-1}| - |T_{n-1}|\},$$

where $R_{n-1} \neq T_{n-1}$ is a Steiner tree for the endpoints of T_{n-1}.

It was proved [5] that a splitting tree is the unique SMT for its endpoints in the Euclidean metric. Now consider the metric induced by the unit sphere, which is a regular hexagon with diagonals parallel to edges of the splitting trees. Since the unit sphere of the Euclidean metric contains that of the hexagonal metric, a tree cannot have greater length in the Euclidean metric than in the hexagonal metric. But splitting trees have identical length in both metrics. Hence a splitting tree must be the unique SMT for its endpoints in the hexagonal metric.

We now consider a subset of splitting trees $\{t_i: i=1,2,\cdots\}$ in the hexagonal metric, where t_i is $T_{3\cdot 2^{i-1}}$ with a symmetrical topology. The topologies of t_1, t_2, and t_3 are shown in Fig. 1. It is clear that the middle Steiner point of t_i, $i \geqq 2$, is not a grid point of $G_1(P)$. Therefore, such t_i are counterexamples for $d=2$.

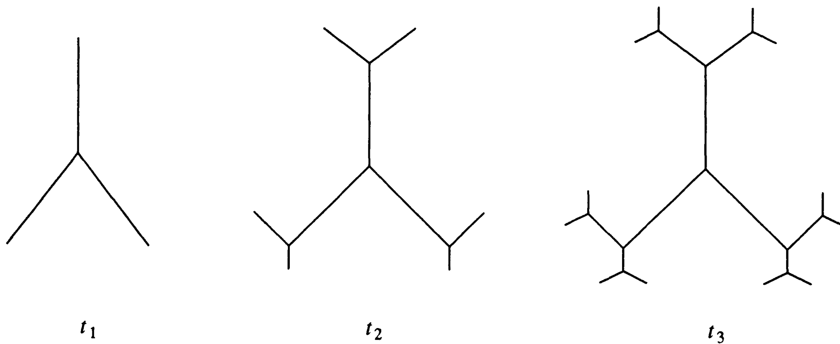


FIG. 1. *Topologies for* t_1, t_2, t_3.

For $d>2$, we construct a unit sphere by using the three directions parallel to the edges of the splitting tree and $d-2$ other orthogonal directions perpendicular to the previous three directions. For the S_d with this unit sphere, the splitting tree is still a unique SMT for its endpoints. As before, there is a Steiner point that is not a grid point. However, the three directions are dependent, which does not meet our condition that every d directions are independent. We will perturb one of the three directions such that our requirement on independence holds.

First, we note that the locations of Steiner points vary continuously as the unit ball varies. Suppose that the minimum Euclidean distance between any grid point and a Steiner point not in the grid is D. Let OA be a diagonal parallel to one direction of the splitting tree. By a continuity argument we can choose a positive number e such that when A varies within Euclidean distance e, every Steiner point varies within Euclidean distance $D/2$. This means that when A varies within Euclidean distance e, the minimum Steiner tree for endpoints of the splitting tree must have a Steiner point not in the grid.

Next, we note that the rest of the d directions other than OA are linearly independent. For any $d-1$ of the d directions, if OA forms a linearly dependent set with them, then A lies in the subspace generated by the $d-1$ directions. However, such finitely many $d-1$ subspaces can not cover a d-dimensional neighborhood of A. So, we can perturb A within Euclidean distance e such that the resulting OA together with the rest d directions induces a unit sphere meeting our requirement.

4. Other grids. For $j=2,3,\cdots$ let $G_j(P)$ be the grid formed by running the $\binom{m}{d-1}$ diagonal hyperplanes through each grid point of $G_{j-1}(P)$. Define a grid point of $G_j(P)$ to be a point where d distinct diagonal hyperplanes intersect. The question is, for

$m > d$, whether there exists an SMT whose Steiner points are all grid points of $G_j(P)$ for some j.

Note that for any fixed j, the middle Steiner point of t_i, $i > j$, is not a grid point in $G_j(P)$. Thus such t_i provide counterexamples for $d = 2$. A perturbation argument extends these counterexamples to $d > 2$ cases. Thus we propose the following stronger conjecture.

CONJECTURE 2. *If $m > d$, then for every fixed j there exists a set P in S_d such that every* SMT *for P contains a Steiner point that is not a grid point of $G_j(P)$.*

Amid these sweeping negative results and conjectures, we show that finite reduction is possible if the unit sphere is a hexagon in the plane such that there exist three unit vectors, parallel to the three diagonals, summing to zero. We denote this space by H^0. If the hexagon is regular, we denote the space by H.

THEOREM 3. *There exists an* SMT *whose Steiner points are all grid points of $G_{n-2}(P)$ if $P \in H^0$ is a set of n points.*

Proof. See the appendix.

For two lines parallel to a diagonal define their distance as the length (in H^0) of the segment lying between the two lines of a line parallel to another diagonal (the choice of which diagonal is immaterial). If P is such that the distance of any two adjacent parallel lines of $G_1(P)$ is a multiple of a constant c, then there exists a k such that $G_k(P)$ is a regular triangular lattice (each side has length c in H^0), and $G_j(P) = G_k(P)$ for all $j > k$.

THEOREM 4. *There exists an* SMT *whose Steiner points are all grid points of $G_k(P)$ if $P \in H^0$ is such that $G_k(P)$ is a regular triangular lattice.*

Proof. The proof for $P \in H$ was given in [3] (the result is a crucial step in proving the Steiner ratio conjecture). Theorem 4 follows by a linear transformation on H.

Extensions of Theorems 3 and 4 to higher dimensions are still open. We state them as conjectures.

Let S_d^0 denote the d-dimensional space, where the unit sphere is a polytope with $d+1$ diagonals such that there exist $d+1$ unit vectors, parallel to the $d+1$ diagonals, summing to zero.

CONJECTURE 3. *There exists an* SMT *whose Steiner points are all grid points of $G_{n-d}(P)$ if $P \in S_d^0$.*

CONJECTURE 4. *There exists an* SMT *whose Steiner points are all grid points of $G_k(P)$ if $P \in S_d^0$ is such that $G_k(P)$ is a lattice of d-dimensional simplices.*

5. Conclusion. The Steiner problem in normed spaces appeared in early literature. For example, Cockayne [1] gave some fundamental properties of SMTs in a normed space whose unit sphere is a symmetric convex surface. Then followed a long gap of void. Recently, there has been a revival of interests on this topic. Du, Graham, and Liu [2] studied the Steiner problem for the plane where the unit disk is a symmetric polygon, and Sarrafzadeh and Wong [6] did it where the unit disk is a regular polygon. In this paper we gave some results that reduce the SMT problem in a normed space to a finite and graphical problem. Our results generalize a recent result of Snyder for rectilinear spaces. We also raised some conjectures as open problems.

Appendix. In H an edge is a path between two vertices (P-points or Steiner points). Thus, an edge can contain several straight segments. An edge is called a *straight edge* if it contains only one straight segment and is called a *nonstraight edge* if it is not a straight edge. Note that an edge with more than two straight segments can always be replaced by an edge with at most two straight segments. Thus, we assume in the following that every edge has at most two straight segments. A Steiner tree for n points

is said to be *full* if it contains exactly $n-2$ Steiner points. Any Steiner tree that is not full can be decomposed into edge-disjoint unions of smaller full Steiner trees, called *full subtrees.*

LEMMA. *There exists an* SMT *in H such that each of its full subtrees contains at most one nonstraight edge.*

Proof. For simplicity, a nonstraight edge is said to be *unexpected* if it is incident to a Steiner point. Consider SMTs with the minimum number of Steiner points. Among them, the tree T with the minimum number of unexpected nonstraight edges will be shown to have the desired property in the lemma. To do so, we show the following two facts.

(1) If two nonstraight edges meet at a Steiner point, we can decrease the number of unexpected nonstraight edges without increasing the number of Steiner points.

(2) If a nonstraight edge and a straight edge meet at a Steiner point with the third edge being straight, then we can decrease the number of Steiner points, or decrease the number of unexpected nonstraight edges without changing the number of Steiner points, or shift the nonstraightness from one edge to another edge without increasing the number of Steiner points and the number of unexpected nonstraight edges.

We first show fact (1). Suppose that two nonstraight edges ACJ and BDJ meet at the Steiner point J. There are four subcases as shown in Figs. 2 and 3. (It may be worth mentioning that by the minimality of the length, the subcases in Fig. 4 or subcases that can be reduced to those in Fig. 4 are impossible.) In Fig. 2, suppose without loss

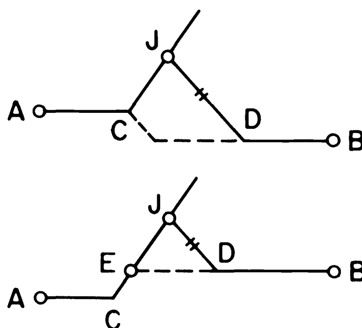


FIG. 2

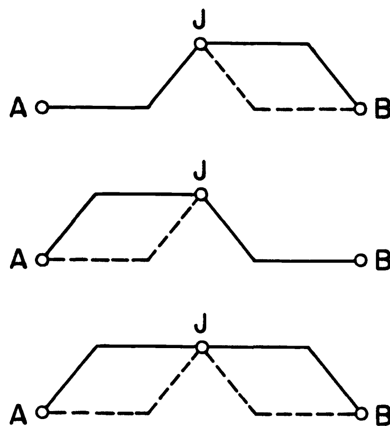


FIG. 3

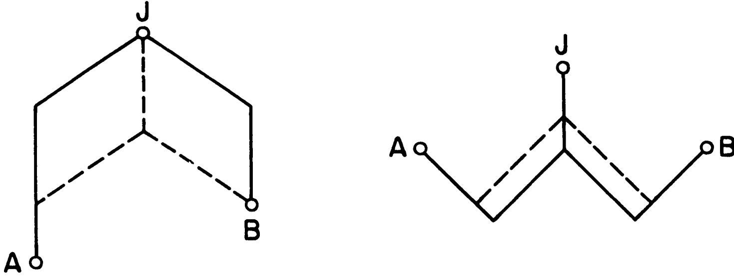


FIG. 4

of generality that the third edge at the Steiner point J meets J in the direction of CJ. If $|JC| \geqq |JD|$, then extending BD to intersect JC at E and deleting JD, we will obtain a new SMT with fewer unexpected nonstraight edges; if $|JC| \leqq |JD|$, then connect C and D, and deleting JD, we can also obtain an SMT with fewer unexpected nonstraight edges. All subcases in Fig. 3 can be reduced to the subcases in Fig. 2. Thus, fact (1) is proved.

Next, we show fact (2). Suppose that the straight edge AJ and the nonstraight edge BDJ meet at the Steiner point J. There are four subcases as shown in Figs. 5, 6, 7, and 8. In Fig. 5, if $|JA| \leqq |JD|$, then we can connect A and D and delete JD so that

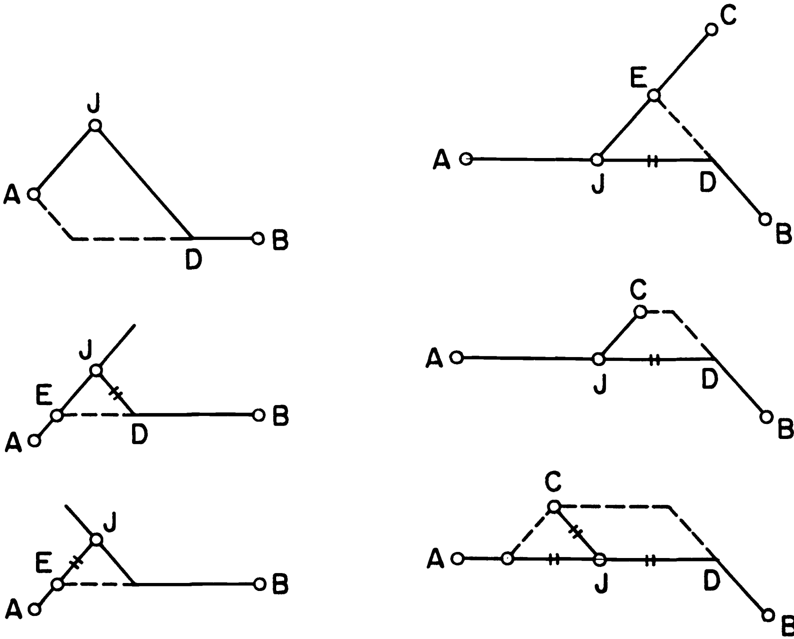


FIG. 5

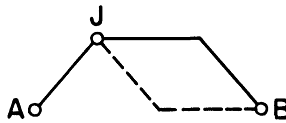


FIG. 6

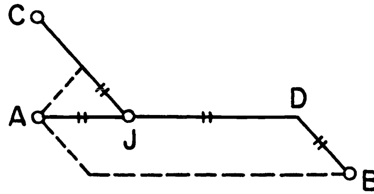


FIG. 7

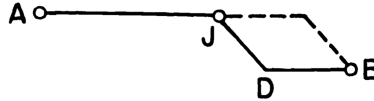


FIG. 8

the resulting tree has one less Steiner point; if $|AJ| \geqq |JD|$, then we extend BD to intersect JA at E and delete either JD or JE depending on the direction of the third edge at J, so that the resulting tree has either fewer unexpected nonstraight edges or a nonstraight edge at A but not at B. The subcases in Figs. 6 and 8 can be reduced to those in Figs. 5 and 7, respectively. How to deal with the subcase in Fig. 7 is shown in the figure. We leave the details to the reader.

Now, by (1), the tree T cannot have a Steiner point incident to two or more nonstraight edges. Furthermore, by (2), in each full subtree of T, we can move all nonstraight edges to meet at the same Steiner point, and hence each full subtree contains at most one nonstraight edge. This completes the proof of the lemma.

We now prove Theorem 3 for $P \in H$. Theorem 3 can then be obtained by a linear transformation on H.

Note that a Steiner point adjacent to two P-points by straight edges appears in $G_1(P)$. In general, a Steiner point adjacent to two grid points of $G_i(P)$ appears as a grid point of $G_{i+1}(P)$. Let T be an SMT with at most one nonstraight edge on each full subtree. Let s be a Steiner point on a full subtree T' with m P-points. Then at least two branches of s contain only straight edges in T', and each $G_i(P)$, $i = 1, 2, \cdots$, contains at least one new Steiner point lying on these two branches until all such Steiner points have been included. Since T' has only $m-2$ Steiner points, $s \in G_{m-2}(P) \subseteq G_{n-2}(P)$.

REFERENCES

[1] E. J. COCKAYNE, *On the Steiner problem*, Canad. Math. Bull., 10 (1967), pp. 431–450.
[2] D. Z. DU, R. L. GRAHAM, AND Z. C. LIU, *Minimum Steiner trees in normed plane*, preprint, 1991.
[3] D. Z. DU AND F. K. HWANG, *A proof of the Gilbert-Pollak conjecture on the Steiner ratio*, Algorithmica, 7 (1992), pp. 121–135.
[4] M. HANAN, *On Steiner problem with rectilinear distance*, SIAM J. Appl. Math., 14 (1966), pp. 255–265.
[5] F. K. HWANG, J. F. WENG, AND D. Z. DU, *A class of full Steiner minimal trees*, Discrete Math., 45 (1983), pp. 107–112.
[6] M. SARRAFZADEH AND C. K. WONG, *Hierarchical Steiner tree construction in uniform orientations*, preprint, May 1991; IEEE Trans. on Computer Aided Designs, to appear.
[7] T. SNYDER, *On the exact location of Steiner points in general dimension*, SIAM J. Comput., 21 (1992), pp. 163–180.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1008-1025, December 1992
© 1992 Society for Industrial and Applied Mathematics
002

ON THE COMPUTATIONAL COMPLEXITY OF APPROXIMATING SOLUTIONS FOR REAL ALGEBRAIC FORMULAE*

JAMES RENEGAR†

Abstract. Upper bounds on the complexity of approximating solutions of general algebraic formulae over the real numbers are established; included are systems of polynomial equations and inequalities.

Key words. polynomials, decision methods, roots

AMS(MOS) subject classifications. 65, 68, 90

1. Introduction[1]. This paper is concerned with the computational complexity of constructing solutions to a very general class of algebraic problems defined over the real numbers. The class includes many nonlinear problems from numerical analysis and mathematical programming. The class is naturally defined in terms of the classical decision problem for the first-order theory of the reals.

The decision problem for the first-order theory of the reals is the problem of determining if expressions of a certain form are true or false. Although a more general form is allowed, all allowable expressions can be reduced to the form

$$(Q_1 x^{[1]} \in \mathbb{R}^{n_1})(Q_2 x^{[2]} \in \mathbb{R}^{n_2}) \cdots (Q_\omega x^{[\omega]} \in \mathbb{R}^{n_\omega}) P(x^{[1]}, \cdots, x^{[\omega]}), \tag{1.1}$$

where

(i) Each Q_k is one of the quantifiers $\exists$ or $\forall$;

(ii) $P(x^{[1]}, \cdots, x^{[\omega]})$ is a quantifier free Boolean formula with *atomic predicates* of the form

$$g_i(x^{[1]}, \cdots, x^{[\omega]}) \Delta_i 0,$$

each $g_i: \bigtimes_{k=1}^{\omega} \mathbb{R}^{n_k} \to \mathbb{R}$ being a real polynomial and Δ_i being any one of the "standard relations"

$$>, \geqq, =, \neq, \leqq, <. \tag{1.2}$$

Such an expression is referred to as a *sentence.* While catenating blocks of variables, it may be assumed that for each k, Q_k and Q_{k+1} are not the same quantifier. Hence, $\omega - 1$ is the number of *quantifier alternations.*

As a simple example of a sentence, consider

$$(\exists y \in \mathbb{R}^n)[(g_1(y) \geqq 0) \wedge \cdots \wedge (g_m(y) \geqq 0)], \tag{1.3}$$

where $g_1, \cdots, g_m: \mathbb{R}^n \to \mathbb{R}$ are polynomials. This sentence asserts that the "feasible set" $\{x: g_i(x) \geqq 0 \ \forall i\}$ is nonempty. Depending on the specific coefficients of the polynomials $g_1, \cdots, g_m$, this sentence is either true or false.

A more interesting example is provided by the following sentence in which $f, g_1, \cdots, g_m: \mathbb{R}^n \to \mathbb{R}$ are assumed to be polynomials:

$$\begin{aligned} &(\exists y \in \mathbb{R}^n)(\forall x \in \mathbb{R}^n)[(g_1(y) \geqq 0) \wedge \cdots \wedge (g_m(y) \geqq 0) \\ &\qquad \wedge \\ &\qquad [(g_1(x) < 0) \vee \cdots \vee (g_m(x) < 0) \vee (f(y) - f(x) \leqq 0)]]. \end{aligned} \tag{1.4}$$

* Received by the editors October 23, 1989; accepted for publication (in revised form) August 15, 1991. This research was supported by National Science Foundation grant DMS-8800835.

† School of Operations Research and Industrial Engineering, College of Engineering, Cornell University, Ithaca, New York 14853.

[1] Through Theorem 1.1, the introduction has much overlap with the introduction of Renegar [19].

This sentence asserts something about the algebraic nonlinear programming problem (NLP)

$$(1.5)\qquad \begin{array}{ll} \min & f(x), \\ \text{s.t.} & g_i(x) \geqq 0, \qquad i=1, \cdots, m. \end{array}$$

The sentence asserts that there exists $y \in \mathbb{R}^n$ such that y is feasible for the NLP and for all $x \in \mathbb{R}^n$, either x is infeasible or the objective function value at y is at least as good as at x. In other words, the sentence asserts that the NLP has an optimal solution y. Depending on the specific coefficients of the polynomials $f, g_1, \cdots, g_m$, this sentence is either true or false.

The collection of all true sentences constitutes the first-order theory of the reals, denoted by Th $(\mathbb{R})$. A decision method for Th $(\mathbb{R})$ is an algorithm that, given a sentence, determines whether sentence is in Th $(\mathbb{R})$. Tarski [23] was the first to present a decision method for Th $(\mathbb{R})$. Regarding computational complexity, much better algorithms than his are now known.

A brief survey of results on the complexity of the decision problem can be found in Renegar [19], [22]. Important results have been established by Collins [8], Grigor'ev [12], and Heintz, Roy, and Solernó [13], among others.

The sentence (1.1) is said to be in prenex form; all quantifiers occur in front. As already mentioned, sentences are allowed to be of a more general form, but all sentences can be reduced to equivalent sentences in prenex form. The reduction can be accomplished efficiently, as is discussed in the introduction of [19]. In the present paper we focus on sentences in prenex form.

Traditionally, attention has been restricted to sentences for which the coefficients of the polynomials g_i are rational numbers. Consequently, a decision method for Th $(\mathbb{R})$ is an algorithm in the usual Turing machine sense. However, there is no ambiguity regarding what is meant for a sentence of the form (1.1) to be true or false if we allow the coefficients of the polynomials g_i to be real numbers. Borrowing a phrase from Blum and Smale [4], we will refer to the resulting collection of true sentences as "the extended first-order theory of the reals" and denote it by ETh $(\mathbb{R})$. Thus, we view Th $(\mathbb{R})$ as the subset of ETh $(\mathbb{R})$, consisting of those sentences for which all of the polynomials occurring in the atomic predicates have rational coefficients.

An appropriate model of computation for defining what is meant by a "decision method for ETh $(\mathbb{R})$" is the model developed by Blum, Shub, and Smale [3]. This model formalizes and extends what researchers often refer to as "arithmetic complexity." Computations are restricted to the arithmetic operations $+, -, \cdot, \div$, all assumed to be performed exactly on real numbers with no rounding errors (i.e., infinite precision), and branching decisions are made using the comparison operations $>$ and $=$. (A complete formalization of the model requires developing an appropriate notion of "uniform algorithm," etc.; these issues are dealt with in [3].)

When speaking of a decision method for Th $(\mathbb{R})$ in the usual Turing machine sense, we will, for brevity, speak of the "bit model" of computation. When speaking of a decision method for ETh $(\mathbb{R})$ as an algorithm in the arithmetic complexity sense, we will speak of the "real number model" of computation. Some discussion of the significance of the real number model of computation as regards the decision problem is presented in [19].

In defining the sentence (1.1), we merely required that P be a "quantifier free Boolean formula." Now we state more precisely the form we will be assuming P to have.

Given an arbitrary Boolean function $\mathbb{P}: \{0, 1\}^m \to \{0, 1\}$ and m atomic predicates $g_i(x)\Delta_i 0$, there is an obvious and natural way to define a 0–1 valued function $P(x)$,

namely,

$$P(x) := \mathbb{P}(B_1(x), \cdots, B_m(x)),$$

where

$$B_i(x) := \begin{cases} 1 & \text{if } g_i(x)\Delta_i 0, \\ 0 & \text{otherwise.} \end{cases}$$

The perspective we take in this paper is that $\mathbb{P}$ is given, and the function P appearing in (1.1) is then defined as above.

In some way a measure of the cost of evaluating the Boolean function $\mathbb{P}$ must enter into the cost of a decision method. Traditionally, $\mathbb{P}$ has been assumed to be of restricted forms. Rather than requiring $\mathbb{P}$ to be of a restricted form, we assume that a procedure (i.e., oracle) is available for evaluating $\mathbb{P}$ when arbitrary values 0 or 1 are substituted for the variables B_i. A component of the bounds we state will be the number of "calls to $\mathbb{P}$," meaning the number of times the procedure for evaluating $\mathbb{P}$ is used. Of course we could restrict $\mathbb{P}$ to be of a specific form, but doing so would reduce the versatility of our results.

When stating time bounds for parallel computation, we will use Time $(\mathbb{P}, N)$ to denote the worst-case time (over all 0–1 vectors) required to compute $\mathbb{P}$ using N processors.

When we refer to "operations" it will be in the context of ETh $(\mathbb{R})$. Formally, for the sequential operation bounds that follow, "operations" can be taken to refer to those allowed in the real number model of computation developed by Blum, Shub, and Smale [3]. For readers unfamiliar with that paper, "operations" can simply be taken to refer to the ordered field operations $+, -, \cdot, \div, >$, and $=$ (and operations for storing and retrieving data). Although a model for parallel computation over the reals is not formalized in [3], the uniform and elementary nature of the algorithms designed for proving the "real number model parallel bounds" that follow guarantee that the bounds will hold for any reasonable real number model of parallel computation.

When we refer to "bit operations" it will be in the context of Th $(\mathbb{R})$ and will refer to Turing machine operations. As with the real number model algorithms, the uniform and elementary nature of the algorithms designed for proving the "bit model parallel bounds" that follow guarantee that the bounds will hold for any reasonable bit model of parallel computation, of which there are several (e.g., the circuit model commonly used in defining NC).

In what follows we assume that $P(x^{[1]}, \cdots, x^{[\omega]})$ has m atomic predicates and we assume that $d \geqq 2$ is an upper bound on the degrees of the polynomials occurring in the atomic predicates. Also recall that n_k is the number of variables occurring in $x^{[k]}$.

When referring to a decision method for Th $(\mathbb{R})$ we may assume that the coefficients of the polynomials are integers; we then let L denote the maximum, over all of the coefficients, of the number of bits required to specify the coefficient.

The data specifying a sentence is $\omega, n_1, \cdots, n_\omega, Q_1, \cdots, Q_\omega, m, \Delta_1, \cdots, \Delta_m, d$ the coefficients of the polynomials $g_1, \cdots, g_m$, and the Boolean function $\mathbb{P}$.

THEOREM 1.1 (Renegar [19], [20]). *There is an algorithm for the decision problem for* ETh $(\mathbb{R})$ *that requires only*

$$(md)^{2^{O(\omega)}\prod_k n_k} \quad \textit{operations and} \quad (md)^{O(\sum_k n_k)} \quad \textit{calls to } \mathbb{P}.$$

The algorithm requires no divisions. The algorithm can be implemented in parallel, requiring time

$$\left[2^\omega \left(\prod_k n_k\right) \log (md)\right]^{O(1)} + \text{Time}(\mathbb{P}, N)$$

if $(md)^{2^{O(\omega)}\prod_k n_k}$ *processors are used for the operations and* $N(md)^{O(\sum_k n_k)}$ *processors are used for the calls (for any* $N \geqq 1$*).*

When restricted to sentences involving only polynomials with integer coefficients, the algorithm becomes a decision method for Th $(\mathbb{R})$ *requiring only*

$$L(\log L)(\log\log L)(md)^{2^{O(\omega)}\prod_k n_k}$$

sequential bit operations and $(md)^{O(\sum_k n_k)}$ *calls to* $\mathbb{P}$. *When implemented in parallel the algorithm requires time*

$$\log(L)\left[2^{\omega}\left(\prod_k n_k\right)\log(md)\right]^{O(1)} + \text{Time}(\mathbb{P}, N)$$

if $L^2(md)^{2^{O(\omega)}\prod_k n_k}$ *processors are used for bit operations and* $N(md)^{O(\sum_k n_k)}$ *processors are used for the calls (for any* $N \geqq 1$*).*

A similar theorem was established by Heintz, Roy, and Solernó [13].

As an application of the theorem, recall the two examples of sentences that we gave. Applying the algorithm of the theorem to the first sentence shows that in the real number model we can determine if the feasible region of the NLP(1.5) is nonempty with $(md)^{O(n)}$ operations performed in time $[n\log(md)]^{O(1)}$ using $(md)^{O(n)}$ parallel processors, assuming that the degrees of the polynomials are at most d. Applying the algorithm to the second sentence shows that we can determine if the NLP has an optimal solution with $(md)^{O(n^2)}$ operations performed in time $[n\log(md)]^{O(1)}$ using $(md)^{O(n^2)}$ parallel processors. The theorem provides analogous bounds for the bit model assuming that the coefficients of $f, g_1, \cdots, g_m$ are integers.

There are many, many interesting decision problems that can be reduced to the problem of deciding if a particular sentence is true or false. Indeed, the importance of the decision problem for the first-order theory of the reals is largely a consequence of its generality and the fact that decision methods for it exist. We list a few more problems, motivated by nonlinear mathematical programming, that the reader may find of interest. With a little practice, the reader can undoubtedly construct many others.

In the examples, we use "$A \Rightarrow B$" as shorthand for "$(\sim A) \vee (A \wedge B)$," and we use "$x$ is feasible" as shorthand for "$(g_1(x) \geqq 0) \wedge \cdots \wedge (g_m(x) \geqq 0)$," i.e., x is feasible for the NLP (1.5).

The following sentence is true if and only if (1.5) has a local optimum:

$$\begin{aligned}(\exists y \in \mathbb{R}^n)(\exists \delta \in \mathbb{R})(\forall x \in \mathbb{R}^n)[(\delta > 0) \wedge (y \text{ is feasible})\\ \wedge\\ ([(x \text{ is feasible}) \wedge (\|x-y\|^2 \leqq \delta^2)] \Rightarrow (f(x)-f(y) \geqq 0))].\end{aligned}$$

Here, $\|\cdot\|$ denotes the Euclidean norm.

For the next example assume that the objective polynomial f is dependent on an additional parameter ρ. Assume that for the situation of interest this parameter is only known to lie within the range $[0, \bar{\rho}]$. The following question is then natural: given $\delta > 0$, does there exist $y \in \mathbb{R}^n$ such that if ρ is fixed at any value in the interval $[0, \bar{\rho}]$, then the resulting NLP (1.5) has an optimal solution within distance δ of y? The answer is yes if and only if the following sentence is true:

$$\begin{aligned}(\exists y \in \mathbb{R}^n)(\forall \rho \in \mathbb{R})(\exists x \in \mathbb{R}^n)(\forall z \in \mathbb{R}^n)[(\rho > 0) \wedge (\rho \leqq \bar{\rho})]\\ \Rightarrow\\ [(x \text{ is feasible}) \wedge (\|y-x\|^2 - \delta^2 \leqq 0)\\ \wedge\\ [(z \text{ is feasible}) \Rightarrow (f(\rho, z) - f(\rho, x) \geqq 0)]].\end{aligned}$$

Similarly, we can construct interesting sentences when the constraint polynomials depend on additional parameters. We can construct a sentence to determine if there is a nondegenerate optimal solution. We can construct sentences to determine which constraints are redundant. We can construct sentences to determine if the feasible set is convex and if the objective and constraint polynomials are convex. We can also do all of these things if the functions involved are only piecewise polynomial, assuming that the underlying decomposition of the domain space is defined via polynomial equalities and inequalities. With only a little thought, we can also do the same things for rational functions (i.e., quotients of polynomials). By introducing additional variables and atomic predicates, we can also introduce radicals into the sentences, e.g., a new variable y and an atomic predicate requiring that $y^2 - x = 0$ allows y to be used as the square root of x. (Of course adding new variables can be disastrous in terms of the complexity bounds provided by Theorem 1.1.) Finally, we remark that deciding if an algebraic "min-max" problem has a solution can often be easily recast into deciding if a particular sentence is true or false, etc.

In the above examples we have been primarily concerned with determining if a solution of an algebraic problem exists. (Is there a solution to the feasibility constraints of the nonlinear programming problem? Does the NLP have an optimal solution?) The present paper is concerned with the computational complexity of approximating such solutions when they do exist.

To be precise, we first introduce the definition of a *formula*. A formula is defined exactly the way a sentence is, except that in a formula not all variables are required to be quantified. The variables that are not quantified are referred to as the *free* variables; when specific values are substituted for the free variables, the formula becomes a sentence.

Consider a formula

$$(Q_1 x^{[1]} \in \mathbb{R}^{n_1}) \cdots (Q_\omega x^{[\omega]} \in \mathbb{R}^{n_\omega}) P(y, x^{[1]}, \cdots, x^{[\omega]}) \tag{1.6}$$

with free variables $y = (y_1, \cdots, y_l)$. We say that $\bar{y} \in \mathbb{R}^l$ is a *solution* for the formula if the sentence obtained by substituting $\bar{y}$ into the formula is true. We say that $\hat{y} \in \mathbb{R}^l$ is an *ε-approximate solution* for the formula if there exists a solution $\bar{y}$ for the formula such that $\|\hat{y} - \bar{y}\| \leqq \varepsilon$, where the norm is the Euclidean norm on $\mathbb{R}^l$. This paper is concerned with the computational complexity of constructing ε-approximate solutions.

As an example, a point $\bar{y}$ is an ε-approximate solution for the quantifier free formula

$$[(g_1(y) \leqq 0) \wedge \cdots \wedge (g_m(y) \leqq 0)]$$

if and only if it is within distance ε of a feasible point for the algebraic NLP (1.5). Similarly, a point $\bar{y}$ is an ε-approximate solution for the formula

$$\begin{gathered}(\forall x \in \mathbb{R}^n)[(g_1(y) \geqq 0) \wedge \cdots \wedge (g_m(y) \geqq 0) \\ \wedge \\ [(g_1(x) < 0) \vee \cdots \vee (g_m(x) < 0) \vee (f(y) - f(x) < 0)]]\end{gathered}$$

if and only if it is within distance ε of an optimal solution of the NLP.

Given a formula (1.6) and $r \geqq 0$, define SOLUTIONS (r) to be the set of all solutions $\bar{y}$ satisfying $\|\bar{y}\| \leq r$. The following theorem is our main result.

THEOREM 1.2. *There are algorithms which, given $0 < \varepsilon < r$ and a formula* (1.6), *construct a set $\{y^{(i)}\}_i$ of $(md)^{2^{O(\omega)} l \prod_k n_k}$ distinct ε-approximate solutions with the property that for each connected components of* SOLUTIONS (r), *at least one of the points, $y^{(i)}$ is within distance ε of the component.*

One such real number model algorithm requires

$$(md)^{2^{O(\omega)} l \prod_k n_k} \log\log\left(3+\frac{r}{\varepsilon}\right)$$

sequential operations and $(md)^{2^{O(\omega)} l \prod_k n_k}$ *calls to* $\mathbb{P}$.

Another such real number model algorithm requires

$$(md)^{2^{O(\omega)} l \prod_k n_k} \log\left(1+\frac{r}{\varepsilon}\right)$$

operations (no divisions), and $(md)^{2^{O(\omega)} l \prod_k n_k}$ *calls to* $\mathbb{P}$; *this algorithm is significant because it can be implemented in parallel, requiring time*

$$\left[2^{\omega} l\left(\prod_k n_k\right) \log(md)\right]^{O(1)} \log\left(1+\frac{r}{\varepsilon}\right) + \text{Time}(\mathbb{P}, N)$$

if $(md)^{2^{O(\omega)} l \prod_k n_k}$ *processors are used for operations and* $N(md)^{2^{O(\omega)} l \prod_k n_k}$ *processors are used for calls (for any* $N \geqq 1$).

Assuming $0 < \varepsilon < r$ *are integral powers of* 2, *there is such a bit model algorithm which, when implemented in parallel, requires time*

$$\left[2^{\omega} l\left(\prod_k n_k\right) \log(mdL + |\log(\varepsilon)| + |\log(r)|)\right]^{O(1)} + \text{Time}(\mathbb{P}, N)$$

if $(L + |\log(\varepsilon)| + |\log(r)|)^{O(1)}(md)^{2^{O(\omega)} l \prod_k n_k}$ *processors are used for operations and* $N(md)^{2^{O(\omega)} l \prod_k n_k}$ *processors are used for calls (for any* $N \geqq 1$). *The* ε*-approximate solutions constructed will then have rational coordinates with numerators and denominators bounded in bit length by* $O(\log(l) + |\log(\varepsilon)| + |\log(r)|)$.

Of course the theorem provides upper bounds on the computational complexity of approximating solutions to the various problems already discussed. We leave determination of the bounds implied to the reader.

The bit model algorithm of the theorem relies heavily on the recent algorithm of Neff [16] for approximating all roots of univariate polynomials. Neff resolved positively the longstanding open problem of whether approximating all roots can be done quickly in parallel.

Neff [16] dealt specifically with bit complexity; however, it seems that slight modifications of his ideas lead to an efficient parallel real number model algorithm for approximating roots of univariate polynomials; namely, given $0 < \varepsilon < r$, it seems that ε-approximations to all roots within distance r of the origin can be obtained in time $\log(d + \log(1 + (r/\varepsilon)))^{O(1)}$ using $(d + \log(1 + (r/\varepsilon)))^{O(1)}$ parallel processors, where d is the degree of the polynomial. If this is indeed true, then Theorem 3.2 implies a corresponding parallel time bound for approximating solutions of general formulae; namely, time

$$\left[2^{\omega} l\left(\prod_k n_k\right) \log\left(md + \log\left(1+\frac{r}{\varepsilon}\right)\right)\right]^{O(1)} + \text{Time}(\mathbb{P}, N)$$

if $(md)^{2^{O(\omega)} l \prod_k n_k}(\log(1 + (r/\varepsilon)))^{O(1)}$ processors are used for operations and $N(md)^{2^{O(\omega)} l \prod_k n_k}$ processors are used for calls (for any $N \geqq 1$). Since a rigorous extension of Neff's algorithm to the real number model computation has not been written, we cannot claim this bound for general formulae to be proven. Moreover, even if true, contrasting this parallel time bound with the sequential time bound for the first algorithm in the theorem leads to the open question of whether there exists a

real number algorithm achieving the same parallel time bound using only $(md)^{2^{O(\omega)l}\prod_k n_k}$ processors for operations and $(md)^{2^{O(\omega)l}\prod_k n_k}N$ processors for calls.

In terms of ε and r alone, the bound provided by the first algorithm of the theorem is optimal for a very general model of computation. More precisely, the following lower bound is known. Let $\mathscr{A}$ denote an algorithm which, given any $s \in [0, r^2]$, constructs a value within distance ε of $\sqrt{s}$. In § 2 of Renegar [17] it is proven that if $\mathscr{A}$ is an algorithm in terms of a very general model of computation that allows the operations $+, -, \cdot, \div, >$, and $=$, then the following is true: there exists $s \in [0, r^2]$ such that when algorithm $\mathscr{A}$ is applied to s, it will require at least $C \log\log(3+(r/\varepsilon))$ operations, where $C>0$ is independent of $\mathscr{A}$, r, and ε. The optimality claim follows.

Theorem 1.2 is somewhat unsatisfactory in that r is given a priori. It would be nice to also have an upper bound on the computational complexity of obtaining a single ε-approximation solution when only ε and the formula (1.6) are input, and not r. The previously mentioned lower bound of [17] implies that an additional parameter must occur in any such upper bound. More specifically, defining

$$r(\varepsilon)=\inf\{r;\ r>\varepsilon \text{ and SOLUTIONS } (r)\neq\emptyset\},$$

the lower bound result implies that in terms of ε and $r(\varepsilon)$ alone, the best upper bound possible (for a very general model of computation) would grow like $\log\log(3+(r(\varepsilon)/\varepsilon))$.

If we could design an algorithm which, given $\varepsilon>0$ and a formula (1.6) for which the solution set is nonempty (a condition that can be efficiently verified using the algorithm of Theorem 1.1), efficiently constructs a good upper bound $\bar{r}(\varepsilon)$ to $r(\varepsilon)$, then combined with the algorithms of Theorem 1.2, we would have methods for efficiently constructing an ε-approximate solution where the only input to the methods would be ε and the formula. However, it is easy to design an algorithm for determining a good upper bound $\bar{r}(\varepsilon)$ to $r(\varepsilon)$ using Theorem 1.1.

First check if SOLUTIONS (ε) is nonempty. If it is nonempty, let $r(\varepsilon):=\varepsilon$. Otherwise, replace ε with 2ε and try again. Assuming that on the ith iteration it is determined that SOLUTIONS $(s_i\varepsilon)=\emptyset$ for a specific number s_i, replace s_i with $s_{i+1}=(s_i)^2$ and try again. Terminate with the first value of s_i thus obtained for which SOLUTIONS $(s_i\varepsilon)\neq\emptyset$, and define $\bar{r}(\varepsilon):=s_i\varepsilon$.

Combining this procedure with the first algorithm of Theorem 1.2 yields a method for constructing an ε-approximate solution with operation count, in terms of ε and $r(\varepsilon)$ alone, growing only like $\log\log(3+(r(\varepsilon)/\varepsilon))$. By the previous remarks, this is optimal in terms of ε and $r(\varepsilon)$ alone (for a very general model of computation), among all algorithms depending only on input ε and the formula (1.6).

The author wishes he knew how to design an efficient real number model algorithm for determining a relatively sharp upper bound on the infimum of those values r for which *every* connected component of the solution set intersects $\{y;\ \|y\|\leqq r\}$. For the bit model of computation, we do have the following proposition, which is established in § 3.

Given a formula (1.6), let SOLUTIONS denote the set of its solutions.

PROPOSITION 1.3. *If formula* (1.6) *has only integer coefficients, each of bit length at most L, then every connected component of* SOLUTIONS *intersects* $\{y;\ \|y\|\leqq r\}$, *where r satisfies* $\log(r)=L(md)^{2^{O(\omega)l}\prod_k n_k}$.

A similar, but weaker bound can be found in Vorobjov [24].

Our proofs are not lengthy because the bulk of the mathematics needed to establish them has already been developed in Renegar [19]-[21]. In § 2 we collect the propositions from those papers that we will rely on. In § 3 we reduce the problem of designing

algorithms to establish Theorem 1.2 to the problem of designing efficient algorithms for approximating zeros of real univariate polynomials; the results in § 3 are phrased to be applicable to any univariate polynomial zero approximation algorithm. In § 4 we recall some known facts regarding the computational complexity of approximating the real zeros of real univariate polynomials. The results of §§ 3 and 4 together give the theorems.

Several researchers have considered the problem of obtaining worst-case computational complexity bounds for approximating solutions of systems of polynomial equations, including Lazard [15], Chistov and Grigor'ev [7], Renegar [18], and Canny [6]. Grigor'ev and Vorobjov [11] considered the problem of approximating solutions of real polynomial inequalities. (Except for [18], the analyses and algorithms in these papers rely on structure provided by the bit model of computation that is not available in the real number model.) When specialized to systems of polynomials, Theorem 1.2 provides bounds at least as good as those obtained by all of these researchers, except for the fact that the constants in the exponent are unspecified.

An understanding of the decision methods of Collins [8], Grigor'ev [12], and Heintz, Roy, and Solernó [13] lead to algorithms for approximating solutions of formulae, similar to the way in which the algorithms in the present paper are developed from [19]-[21]. (Both Collins and Grigor'ev deal only with bit complexity and do not present efficient parallel decision methods.) However, in the same ways that the complexity bounds in [19] are superior to those found in these other works (see [19] or [22] for a comparison), the resulting bounds for approximating solutions are also superior.

Finally, as will become obvious to anyone who proceeds, this is strictly a theoretical work. Although the ideas underlying the algorithms may someday lead to "practical" algorithms, the algorithms herein are constructed solely as means to proving the theorems.

2. Preliminaries. In this section we introduce definitions and record several previously established propositions.

The notation used in this and subsequent sections may strike the reader as odd; it has been chosen to conform with the notation of [19]-[21], where the reader is referred for many of the proofs.

Whenever we speak of "constructing" something, we mean that there is a real number model algorithm for doing so. Each of the algorithms in the following propositions and lemmas yield bit model algorithms when restricted to integer inputs, assuming that the underlying operations are carried out "bit by bit."

Let $h_1, \cdots, h_{\mathcal{M}}: \mathbb{R}^l \to \mathbb{R}$ be arbitrary polynomials of degree at most $\mathcal{D}$. We use $\{h_i\}_i$ to denote the set of these polynomials. A vector $\sigma \in \{-1, 0, 1\}^{\mathcal{M}}$ is said to be a "consistent sign vector" for $\{h_i\}_i$ if there exists $\bar{y} \in \mathbb{R}^l$ such that the sign of $h_i(\bar{y})$ is σ_i for all i. The "sign vector of $\{h_i\}_i$ at $\bar{y}$" is the vector in $\{-1, 0, 1\}^{\mathcal{M}}$ whose ith coordinate has the same sign as $h_i(\bar{y})$.

The following proposition is a restatement of Proposition 4.1 from [19].

PROPOSITION 2.1. *Any set $\{h_i\}_i$ of $\mathcal{M}$ polynomials $h_i: \mathbb{R}^l \to \mathbb{R}$, of degree at most $\mathcal{D} \geqq 2$, has at most $(\mathcal{M}\mathcal{D})^{O(l)}$ consistent sign vectors. The entire set of consistent sign vectors can be constructed from the coefficients of $\{h_i\}_i$ with $(\mathcal{M}\mathcal{D})^{O(l)}$ operations (no divisions) performed in time $[l \log(\mathcal{M}\mathcal{D})]^{O(1)}$ using $(\mathcal{M}\mathcal{D})^{O(l)}$ parallel processors. If the coefficients of $\{h_i\}_i$ are integers of bit length at most L, the construction can be accomplished with $L(\log L)(\log\log L)(\mathcal{M}\mathcal{D})^{O(l)}$ sequential bit operations, or in time $(\log L)[l \log(\mathcal{M}\mathcal{D})]^{O(1)}$ using $L^2(\mathcal{M}\mathcal{D})^{O(l)}$ parallel processors.*

The "connected sign partition" $\text{CSP}\{h_i\}_i$ generated by a finite set $\{h_i\}_i$ of polynomials $h_i: \mathbb{R}^l \to \mathbb{R}$ is the partition of $\mathbb{R}^l$ whose elements are the maximal connected subsets with the following property: if $\bar{y}$ and $\hat{y}$ are in the same element, then the sign of $h_i(\bar{y})$ is the same as the sign $h_i(\hat{y})$ for all i.

The following proposition is a restatement of Proposition 6.2.2 from [20]. The polynomials $\{g_i\}_i$ occurring in the proposition are assumed to be those occurring in the formula (1.6).

PROPOSITION 2.2. *Given a formula* (1.6), *there exists a set* $\{h_i\}_i$ *of* $(md)^{2^{O(\omega)}\prod_k n_k}$ *polynomials* $h_i: \mathbb{R}^l \to \mathbb{R}$, *of degree at most* $(md)^{2^{O(\omega)}\prod_k n_k}$, *with the property that if* $\bar{y}$ *and* $\hat{y}$ *are in the same element of* $\text{CSP}\{h_i\}_i$, *then* $\bar{y} \in \text{SOLUTIONS}$ *if and only if* $\hat{y} \in \text{SOLUTIONS}$. *The set* $\{h_i\}_i$ *can be constructed from the coefficients of* $\{g_i\}_i$ *with* $(md)^{2^{O(\omega)}l\prod_k n_k}$ *operations (no divisions) in time* $[2^\omega l(\prod_k n_k)\log(md)]^{O(1)}$ *using* $(md)^{2^{O(\omega)}l\prod_k n_k}$ *parallel processors. If the coefficients of* $\{g_i\}_i$ *are integers of bit length at most L, then the construction can be accomplished with* $L(\log L)(\log\log L)(md)^{2^{O(\omega)}l\prod_k n_k}$ *sequential bit operations, or in time* $(\log L)[2^\omega l(\prod_k n_k)\log(md)]^{O(1)}$ *if* $L^2(md)^{2^{O(\omega)}l\prod_k n_k}$ *parallel processors are used; moreover, the coefficients of* $\{h_i\}_i$ *will then be integers of bit length at most* $(L+l)(md)^{2^{O(\omega)}\prod_k n_k}$.

For ξ, $U \in \mathbb{C}^{l+1}$ define $\xi \cdot U = \sum_j \xi_j U_j$. If ξ satisfies $\xi_{l+1} \neq 0$, define

$$\text{Aff}(\xi) := \frac{1}{\xi_{l+1}}(\xi_1, \cdots \xi_l) \in \mathbb{C}^l,$$

the "affine image" of ξ. Let

$$e_{l+1} := (0, \cdots, 0, 1) \in \mathbb{R}^{l+1}.$$

For a polynomial $R: \mathbb{R}^{l+1} \to \mathbb{R}$ in the variables $U_1, \cdots, U_{l+1}$, define

$$\nabla R := \left(\frac{\partial R}{\partial U_1}, \cdots, \frac{\partial R}{\partial U_{l+1}}\right).$$

The following proposition is a partial restatement of Proposition 3.8.1 from [19].

PROPOSITION 2.3. *Assume that* $h_1, \cdots, h_{\mathcal{M}}: \mathbb{R}^l \to \mathbb{R}$ *are polynomials of degree at most* $\mathcal{D} \geqq 2$. *There exists a set* $\mathcal{R}\{h_i\}_i$ *of* $(\mathcal{M}\mathcal{D})^{O(l)}$ *polynomials* $R: \mathbb{R}^{l+1} \to \mathbb{R}$ *of degree at most* $D = (\mathcal{M}\mathcal{D})^{O(l)}$ *with the following properties*:

(i) *For each element of* $\text{CSP}\{h_i\}_i$ *there exists* $R \in \mathcal{R}\{h_i\}_i$ *such that R is not identically zero and factors linearly over the complexes* $R(U) = \prod_i \xi^{(i)} \cdot U$, *where for some i*, $\text{Aff}(\xi^{(i)})$ *is well defined and is in the element*;

(ii) *For each* $\beta \in \mathbb{R}^{l+1}$ *the entire set of univariate polynomials*

$$t \mapsto R(\beta + te_{l+1}),$$

$$t \mapsto \frac{d^j}{dt^j}\nabla R(\beta + te_{l+1}), \qquad j = 0, \cdots, D$$

obtained from all $R \in \mathcal{R}\{h_i\}_i$ *can be constructed from* β *and the coefficients of* $\{h_i\}_i$ *with* $(\mathcal{M}\mathcal{D})^{O(l)}$ *operations (no divisions) in time* $[l\log(\mathcal{M}\mathcal{D})]^{O(1)}$ *using* $(\mathcal{M}\mathcal{D})^{O(l)}$ *parallel processors; if the coefficients of* $\{h_i\}_i$ *and* β *are integers of bit length at most L, then all numbers occurring during the construction will be integers of bit length at most* $L(\mathcal{M}\mathcal{D})^{O(l)}$.

The significance of the bound on the bit length of the integers occurring during the construction is that bit operation bounds are easily deduced from it and the real number model operation bounds of the proposition; this is made especially easy because the construction avoids divisions. For example, because two integers of bit length at most L can be multiplied in sequential time $O(L(\log L)(\log\log L))$, the

proposition gives an overall sequential bit operation bound of $L(\log L)(\log\log L)(\mathcal{M}\mathcal{D})^{O(l)}$. Similarly, a parallel time bound for the bit model is easily deduced from the fact that two integers of bit length at most L can be multiplied in time $O(\log L)$ using $O(L^2)$ parallel processors.

Define

$$\mathcal{B}(l+1, D) := \{(i^{l-1}, i^{l-2}, \cdots, i, 1, 0);\ i \in \mathbb{Z} \text{ and } 0 \leqq i \leqq lD^2\}.$$

Thus, $\mathcal{B}(l+1, D) \subset \mathbb{R}^{l+1}$.

The next proposition is a partial restatement of Proposition 2.3.1 from [19].

PROPOSITION 2.4. *Given any real polynomial $R: \mathbb{R}^{l+1} \to \mathbb{R}$ of degree at most D that is not identically zero and factors linearly over the complex numbers $R(U) = \prod_i \xi^{(i)} \cdot U$, the following is true: for each $\xi^{(i)}$ for which* Aff $(\xi^{(i)})$ *is well defined and real there exist $\beta \in \mathcal{B}(l+1, D)$ and $0 \leqq k \leqq D$ such that the univariate polynomial $t \mapsto R(\beta + te_{l+1})$ is not identically zero, and for some real zero $\bar{t}$ of $t \mapsto R(\beta + te_{l+1})$, the vector*

$$\bar{\xi} := \frac{d^k}{dt^k} \nabla R(\beta + \bar{t} e_{l+1})$$

satisfies $\mathrm{Aff}\,(\bar{\xi}) = \mathrm{Aff}\,(\xi^{(i)})$.

The following easily proven proposition is a restatement of Proposition 4.1.1 of [20]. The importance of this proposition has been recognized by others (e.g., see Coste and Roy [10]).

PROPOSITION 2.5 (Thom's lemma). *Assume that $p \not\equiv 0$ is a real univariate polynomial of degree d. If $t', t'' \in \mathbb{R}$ are such that $t' < t''$ and for some $0 \leqq i < d$ there is a real zero of the* ith *derivative $p^{(i)}$ contained in the interval $[t', t'']$, then for some $i \leqq j < d$ the sign of $p^{(j)}(t')$ differs from the sign of $p^{(j)}(t'')$.*

As a simple consequence of the proposition, note that if $p \not\equiv 0$, $t' \neq t''$, $p(t') = p(t'') = 0$, then the sign vector of $\{p^{(j)}\}_{j=0}^{d-1}$ at t' differs from that at t''. Hence, the sign vectors of $\{p^{(j)}\}_{j=0}^{d-1}$ at the real zeros of p serve as representatives of the zeros; the sign vectors distinguish the zeros from one another.

Let $p(t) = \sum_{i=0}^{d} a_i t^i$, $q(t) = \sum_{i=0}^{e} b_i t^i$ be univariate polynomials of degrees at most d and e, respectively. The "Sylvester resultant" of p and q is the determinant of the $(d+e) \times (d+e)$ "Sylvester matrix" $[m_{ij}]$ defined by

$$m_{ij} := \begin{cases} a_{d+j-i} & \text{if } j \leqq e-k, \\ b_{j-i} & \text{if } j > e-k. \end{cases}$$

An extremely well-known and classical result states that if the degrees of p and q are exactly d and e, then the Sylvester resultant of p and q is zero if and only if p and q have a common zero (among the complex numbers). A proof of this is provided by Lemma 3.1 of [20].

Another well-known fact that we will rely on is that interpolation of a univariate polynomial $p(t) = \sum_{i=0}^{d} a_i t^i$ can be accomplished quickly in parallel. A proof of the following easy lemma can be found in Appendix B of [19].

LEMMA 2.6. *Assume that $p: \mathbb{C} \to \mathbb{C}$ is a polynomial of degree at most $d \geqq 2$. A positive multiple of p can be computed solely from the values $p(\bar{t})$, $\bar{t} \in \{0, 1, \cdots, d\}$, using $d^{O(1)}$ operations (no divisions). The computations can be implemented in parallel, requiring time $[\log(d)]^{O(1)}$ if $d^{O(1)}$ processors are used. If the values $p(\bar{t})$, $\bar{t} \in \{0, 1, \cdots, d\}$ are all integers of bit length at most L, all numbers occurring during the computations will be integers of bit length at most $L + d^{O(1)}$.*

Yet another well-known fact that we will use is that the determinant of a matrix can be computed quickly in parallel. The algorithm underlying the following proposition is constructed by slightly extending ideas of Csanky [9] to avoid divisions. A proof of the proposition can be found in Appendix A of [19].

PROPOSITION 2.7 (Csanky [9], Berkowitz [1]). *There exists an algorithm which, given any* $n \geqq 1$ *and any complex* $n \times n$ *matrix* A, *computes* $n! \det(A)$ *without divisions in time* $O(\log^2(n))$ *using* $n^{O(1)}$ *parallel processors. If the coefficients of* A *are integers of bit length at most* L, *all numbers occurring during the computation will be integers of bit length at most* $Ln^{O(1)}$.

Propositions similar to the following proposition are well known (e.g., Borodin, von zur Gathen, and Hopcroft [2]).

PROPOSITION 2.8. *Suppose that* p_1, p_2, *and* p_3 *are real univariate polynomials of degree at most* d. *Let* p *denote the greatest common divisor of* $\{p_1, p_2, p_3\}$. (*Of course* p *is unique up to a constant multiple.*) *Then we can efficiently construct real polynomials* $\bar{p}_1$, $\bar{p}_2$, *and* $\bar{p}_3$ *for which there exists a common constant* $c \neq 0$ *satisfying* $cp_i = p\bar{p}_i$ *for all* i. *By "efficiently construct" we mean that the polynomials* $\bar{p}_i$ *can be constructed with* $d^{O(1)}$ *operations* (*no divisions*) *in time* $[\log(d)]^{O(1)}$ *using* $d^{O(1)}$ *parallel processors. If the coefficients of* p_1, p_2, *and* p_3 *are integers of bit length at most* L, *then all numbers occurring during the construction will be integers of bit length at most* $Ld^{O(1)}$.

Proof. Begin by computing p. It is well known that this can be accomplished in parallel time $[\log(d)]^{O(1)}$ using $d^{O(1)}$ processors (e.g., by relying on Brown and Traub [5] and Csanky [9]). A complete proof of this is provided just following Proposition 8.2 in [21]. The proof there shows that this can be accomplished without divisions by relying on Proposition 2.7 above. Moreover, the proof shows that if p_1, p_2, and p_3 have integer coefficients of bit length at most L, then the constructed polynomial p will have integer coefficients of bit length at most $Ld^{O(1)}$.

To construct $\bar{p}_i$, consider the linear equations corresponding to the identity $p_i = p\hat{p}_i$, viewing the coefficients of $\hat{p}_i$ as variables. Use the algorithm of Proposition 2.7 to efficiently compute (multiples of) the numerator and denominator determinants arising from Cramer's rule. Multiply the quotients by the product of the three denominator determinants (for $i = 1$, 2, and 3) to obtain $\bar{p}_i$. □

We close this section with a well-known and easily proven lemma that will be relied upon in establishing Proposition 1.3.

LEMMA 2.9. *Suppose that* $p(t) = \sum_{i=0}^{d} a_i t^i$ *is a univariate polynomial, where* $a_d \neq 0$. *If* $p(\bar{t}) = 0$, *then* $|\bar{t}| \leqq 1 + \max_{i<d} |a_i/a_d|$.

Proof. We may assume that $|\bar{t}| > 1$. Clearly,

$$|\bar{t}|^d \leqq \sum_{i=0}^{d-1} \left|\frac{a_i}{a_d}\right| |\bar{t}|^i \leqq \max_{i<d} \left|\frac{a_i}{a_d}\right| \sum_{i=0}^{d-1} |\bar{t}|^i < \max_{i<d} \left|\frac{a_i}{a_d}\right| \frac{|\bar{t}|^d}{|\bar{t}|-1}.$$

The lemma follows. □

3. Reduction to univariate polynomial zero approximation. In this section we show how to reduce the problem of constructing algorithms for establishing Theorem 1.2 to the problem of constructing efficient algorithms for approximating zeros of univariate polynomials. The highlight of this section is a theorem that allows us to deduce operation time bounds on the cost of obtaining ε-approximate solutions from operation and time bounds for univariate polynomial zero approximation algorithms.

Before stating and proving the theorem of this section, we present a proposition that will be used in the proof. This proposition regards the computational complexity of approximating the real factors of multivariate polynomials that are known to factor linearly over the complex numbers.

We assume that a real number model procedure for obtaining approximations to the real zeros of real univariate polynomials is available. We treat the procedure as an oracle. Letting P_d denote the set of nonconstant real univariate polynomials of degree at most d, we let Cost (d, r, ε) denote the worst-case (over P_d) number of sequential operations required by the procedure to construct a set of $d^{O(1)}$ points that contains ε-approximations to all of those real zeros $\bar{x}$ satisfying $|\bar{x}| \leqq r$. (We don't require that each of the $d^{O(1)}$ points be an ε-approximation to a zero.) We let Time (d, r, ε, N) denote the worst-case time required by the procedure if it is implemented using N parallel processors. We also assume that we have an analogous bit model procedure. Letting $P_{L,d}$ denote the set of nonconstant real univariate polynomials, of degree at most d, whose coefficients are integers of bit length at most L, we let Cost (L, d, r, ε) denote the worst case number of sequential bit operations required by the procedure, and we let Time $(L, d, r, \varepsilon, N)$ denote the worst case time required by the procedure if it is implemented using N parallel processors. We assume that this procedure constructs rational points $y^{(i)}$ with numerator and denominator bounded in bit length by $O(|\log(r)| + |\log(\varepsilon)|)$.

Recall that $\mathscr{B}(l+1, D) := \{(i^{l-2}, i^{l-2}, \cdots, i, 1, 0);\, i \in \mathbb{Z}, 0 \leqq i \leqq lD^2\}$.

PROPOSITION 3.1. *Assume that $R: \mathbb{R}^{l+1} \to \mathbb{R}$ is a (not identically zero) polynomial, of degree at most $D \geqq 2$, that factors linearly $R(U) = \prod_i \xi^{(i)} \cdot U$ over the complex numbers. Assume that the coefficients of all of the following pairs of univariate polynomials are available*;

$$t \mapsto R(\beta + te_{l+1}), \tag{3.1}$$

$$t \mapsto \nabla R(\beta + te_{l+1}), \tag{3.2}$$

where β ranges over $\mathscr{B}(l+1, D)$. (Note that we do not assume the coefficients of R are known; we do assume that for each β we know which pair (3.1), (3.2) *corresponds to β.)*

Given $0 < \varepsilon < r$, a set $\{y^{(i)}\}_i \subset \mathbb{R}^l$ of $D^{O(1)}$ points satisfying the following property can be efficiently constructed; for each $\xi^{(i)}$ satisfying $\xi^{(i)}_{l+1} \neq 0$, $\mathrm{Aff}(\xi^{(i)}) \in \mathbb{R}^l$, and $\|\mathrm{Aff}(\xi^{(i)})\| \leqq r$, there exists $y^{(j)} \in \{y^{(i)}\}_i$ satisfying $\|y^{(j)} - \mathrm{Aff}(\xi^{(i)})\| \leqq \varepsilon$.

By "efficiently constructed" we mean that the set $\{y^{(i)}\}_i$ can be constructed with $(lD)^{O(1)}$ Cost $(D, r, \varepsilon/l)$ *sequential operations, or in time $[\log(lD)]^{O(1)} +$* Time $(D, r, \varepsilon/l, N)$ *using $N(lD)^{O(1)}$ parallel processors (for any $N \geqq 1$). If the coefficients of the polynomials* (3.1) *and* (3.2) *are all integers of bit length at most L, it can then be constructed with*

$$\left[\hat{L}(\log \hat{L})(\log\log \hat{L}) + \mathrm{Cost}\left(\bar{L}, D, r, \frac{\varepsilon}{l}\right)\right](lD)^{O(1)}$$

sequential bit operations, where $\hat{L} = L + \log(l) + |\log(r)| + |\log(\varepsilon)|$ and $\bar{L} = LD^{O(1)}$. It can then be constructed in parallel, requiring time

$$(\log \hat{L})[\log(lD)]^{O(1)} + \mathrm{Time}\left(\bar{L}, D, r, \frac{\varepsilon}{l}, N\right)$$

if $(\hat{L}^2 + N)(lD)^{O(1)}$ processors are used (for any $N \geqq 1$). The coordinates of the points $y^{(i)}$ will then be rational with numerator and denominator bounded in bit length by $O(\log(l) + |\log(r)| + |\log(\varepsilon)|)$.

Proof. By Proposition 2.4 there exist $0 \leqq k \leqq D$ and $\beta \in \mathscr{B}(l+1, D)$, which satisfy

$$t \mapsto R(\beta + te_{l+1}) \not\equiv 0, \tag{3.3}$$

$$t \mapsto \frac{d^k}{dt^k} \frac{\partial}{\partial U_{l+1}} R(\beta + te_{l+1}) \not\equiv 0. \tag{3.4}$$

Moreover, such k and β can be efficiently determined from the polynomials (3.1), (3.2). Fix such a pair k, β.

Consider the system of $D+1$ univariate polynomials

$$t \mapsto \frac{d^w}{dt^w} R(\beta + te_{l+1}) \qquad w = 0, \cdots, D-1,$$

$$t \mapsto \frac{d^k}{dt^k} \frac{\partial}{\partial U_{l+1}} R(\beta + te_{l+1}).$$

Assuming that the coordinates of the consistent sign vectors for this system are indexed from 1 to $D+1$, define $\mathcal{T}$ as the set consisting of those consistent sign vectors τ for the system that satisfy both of the properties $\tau_1 = 0$, $\tau_{D+1} \neq 0$. Because the coefficients of the polynomials (3.1) and (3.2) are assumed available, Proposition 2.1 shows that $\mathcal{T}$ can be constructed efficiently.

Relying on Proposition 2.5, there is a natural one-to-one correspondence between $\mathcal{T}$ and the set consisting of those real zeros $\bar{t}$ of $t \mapsto R(\beta + te_{l+1})$ for which $\mathrm{Aff}\,[(d^k/dt^k)\nabla R(\beta + \bar{t}e_{l+1})]$ is well defined. For $\tau \in \mathcal{T}$, define $t(\tau)$ to be the zero corresponding to τ, and define

$$y(\tau) := \mathrm{Aff}\left[\frac{d^k}{dt^k} \nabla R(\beta + t(\tau)e_{l+1})\right].$$

We will show how to efficiently construct a finite subset of $\mathbb{R}^l$ that contains an ε-approximation to every point $y(\tau)$ satisfying $\|y(\tau)\| \leqq r$. Letting $\{y^{(i)}\}_i$ denote the union of the subsets thus obtained from all $0 \leqq k \leqq D$ and $\beta \in \mathcal{B}(l+1, D)$ satisfying (3.3) and (3.4), Proposition 2.4 then implies that $\{y^{(i)}\}_i$ satisfies the requirements of the proposition.

We continue to assume that k and β are fixed and that they satisfy (3.3) and (3.4).

For each $i = 1, \cdots, l$, consider the Sylvester resultant of the two univariate polynomials

$$t \mapsto R(\beta + te_{l+1}), \tag{3.5}$$

$$t \mapsto \frac{d^k}{dt^k} \frac{\partial}{\partial U_i} R(\beta + te_{l+1}) - s\frac{d^k}{dt^k} \frac{\partial}{\partial U_{l+1}} R(\beta + te_{l+1}), \tag{3.6}$$

treating the second polynomial as a polynomial of formal degree e, where e is the maximal degree achieved by the second polynomial (as a polynomial in t) as s ranges over $\mathbb{C}$. We can thus view the Sylvester resultant as a real univariate polynomial in the variables s; let q_i denote this polynomial.

From the coefficients of the polynomials (3.1) and (3.2), a real nonzero constant multiple of q_i can be computed quickly in parallel by using the algorithm of Proposition 2.7 to evaluate the determinant of the Sylvester matrix for particular values of s, and then interpolating using the algorithm of Lemma 2.6.

Note that $y_i(\tau)$ is a zero of q_i for all i. This property of q_i will be especially important for us. However, the above construction can produce q_i that are identically zero; this we must avoid. We now modify the construction to produce $q_i \neq 0$, which still has $y_i(\tau)$ as a zero for all τ.

Rewrite the univariate polynomials (3.5) and (3.6) as $t \mapsto p_1(t)$ and $t \mapsto p_2(t) - sp_3(t)$, respectively. Using the algorithm of Proposition 2.8, replace p_1, p_2, and p_3 with the polynomials $\bar{p}_1$, $\bar{p}_2$, and $\bar{p}_3$ occurring in the statement of that proposition. Let $q_i(s)$ denote the Sylvester resultant arising from the polynomials $t \mapsto \bar{p}_1(t)$, $t \mapsto \bar{p}_2(t) - s\bar{p}_3(t)$.

Since $\bar{p}_1$, $\bar{p}_2$, and $\bar{p}_3$ share no common zero, $q_i \neq 0$. Since $p_3(t(\tau)) \neq 0 = p_1(t(\tau))$ for all $\tau \in \mathcal{T}$, we have that $\bar{p}_1(t(\tau)) = 0$ for all $t \in \mathcal{T}$. Consequently, since $p_2/p_3 = \bar{p}_2/\bar{p}_3$, we have that $y_i(\tau)$ is a zero of q_i for all $\tau \in \mathcal{T}$.

Henceforth, we may assume that $q_i \neq 0$ and $q_i(y_i(\tau)) = 0$ for all $\tau \in \mathcal{T}$.

For each $i = 1, \cdots, l$, apply the algorithm for approximating zeros of univariate polynomials to q_i to obtain a set of $D^{O(1)}$ points $\{s_{ij}\}_j \subset \mathbb{R}$ with the property that for each real zero $\bar{s}$ of q_i that satisfies $|\bar{s}| \leqq r$, there exists j such that $|s_{ij} - \bar{s}| \leqq \varepsilon/l$. In particular, for each pair (i, τ), where $\|y(\tau)\| \leqq r$, there exists $j' = j(i, \tau)$ satisfying $|y_i(\tau) - s_{ij'}| \leqq \varepsilon/l$.

We will discuss a procedure that, given i, j, and $\tau \in \mathcal{T}$, efficiently determines if $|y_i(\tau) - s_{ij}| \leqq \varepsilon/l$. Applying this procedure to all triples (i, j, τ), we can then efficiently determine the set $\mathcal{T}^*$ of those τ with the property that for each i there exists $j' = j(i, \tau)$ satisfying $|y_i(\tau) - s_{ij'}| \leqq \varepsilon/l$; of course, we determine indices $j(i, \tau)$ as well. (If there are several such indices for some pair i and τ, discard all but one of them and denote it by $j(i, \tau)$.) Then the set of points $\{(s_{1,j(1,\tau)}, \cdots, s_{l,j(l,\tau)});\ \tau \in \mathcal{T}^*\}$ will contain an ε-approximation to each point $y(\tau)$ $(\tau \in \mathcal{T})$ satisfying $\|y(\tau)\| \leqq r$.

Finally, to complete the proof, here is the procedure that, given i, j, and $\tau \in \mathcal{T}$, efficiently determines if $|y_i(\tau) - s_{ij}| \leqq \varepsilon/l$. Consider the following system of $D+3$ bivariate polynomials:

$$
\begin{aligned}
t &\mapsto \frac{d^w}{dt^w} \nabla R(\beta + te_{l+1}) \qquad w = 0, \cdots, D-1, \\
t &\mapsto \frac{d^k}{dt^k} \frac{\partial}{\partial U_{l+1}} R(\beta + te_{l+1}), \\
(s, t) &\mapsto \frac{d^k}{dt^k} \frac{\partial}{\partial U_i} R(\beta + te_{l+1}) - s \frac{d^k}{dt^k} \frac{\partial}{\partial U_{l+1}} R(\beta + te_{l+1}), \\
s &\mapsto \frac{\varepsilon^2}{l^2} - (s - s_{ij})^2.
\end{aligned} \tag{3.7}
$$

Relying on the algorithm of Proposition 2.1, the set of consistent sign vectors for this system can be efficiently constructed. Using Proposition 2.5, the point $t = t(\tau)$, $s = y_i(\tau)$ is the unique point at which the sign vector of the system has $(\tau, 0)$ as its first $D+2$ coordinates. Searching through the consistent sign vectors, we can thus find the sign vector corresponding to $t = t(\tau)$, $s = y_i(\tau)$; the last coordinate of that sign vector is 0 or 1 if and only if $|y_i(\tau) - s_{ij}| \leqq \varepsilon/l$.

The operation and time bounds of the proposition follows easily from the propositions and lemma referred to in the construction. □

Before continuing, we note the following. If the coefficients of the polynomials (3.1) and (3.2) are integers of bit length at most L, then the coefficients of the polynomial q_i occurring in the proof of the proposition will be integers of bit length at most $LD^{O(1)}$. In particular, since $y_i(\tau)$ (as in the proof) is a zero of q_i for *all* $\tau \in \mathcal{T}$, it follows from Lemma 2.9 that $\log(\|\mathrm{Aff}(\xi^{(i)})\|) \leqq \log(l) + LD^{O(1)}$ for all i such that $\mathrm{Aff}(\xi^{(i)})$ is well defined. This fact will be important in establishing Proposition 1.3.

Let Cost (d, r, ε), Time (d, r, ε, N), Cost (L, d, r, ε), and Time $(L, d, r, \varepsilon, N)$ be as defined just prior to Proposition 3.1. We can now easily establish the following theorem.

THEOREM 3.2. *Given* $0 < \varepsilon < r$ *and a formula* (1.6), *a set* $\{y^{(i)}\}_i$ *of* $(md)^{2^{O(\omega)} l \prod_k n_k}$ *distinct* ε*-approximate solutions satisfying the following property can be efficiently constructed: for each connected component of* SOLUTIONS (r) *there exists* $y^{(i)}$ *within distance* ε *of the component.*

By "efficiently constructed," we mean that the set $\{y^{(i)}\}_i$ *can be constructed with* $(md)^{2^{O(\omega)}l\prod_k n_k}\,\mathrm{Cost}\,(D, r, \varepsilon/l)$ *operations and* $(md)^{2^{O(\omega)}l\prod_k n_k}$ *calls to* $\mathbb{P}$*, where* $D = (md)^{2^{O(\omega)}l\prod_k n_k}$*. It can be constructed in time*

$$\left[2^{\omega}l\left(\prod_k n_k\right)\log(md)\right]^{O(1)} + \mathrm{Time}\left(D, r, \frac{\varepsilon}{l}, N_1\right) + \mathrm{Time}\,(\mathbb{P}, N_2)$$

if $N_1(md)^{2^{O(\omega)}l\prod_k n_k}$ *parallel processors are used for operations and* $N_2(md)^{2^{O(\omega)}l\prod_k n_k}$ *parallel processors are used for calls (for any* $N_1, N_2 \geqq 1$*).*

If the formula has only integer coefficients of bit length at most L, then the set $\{y^{(i)}\}_i$ can be constructed with

$$\left[\hat{L}(\log\hat{L})(\log\log\hat{L}) + \mathrm{Cost}\left(\bar{L}, D, r, \frac{\varepsilon}{l}\right)\right](md)^{2^{O(\omega)}l\prod_k n_k}$$

sequential bit operations, and $(md)^{2^{O(\omega)}l\prod_k n_k}$ calls to $\mathbb{P}$, where $\hat{L} = L + \log(l) + |\log(\varepsilon)| + |\log(r)|$ and $\bar{L} = LD^{O(1)}$. It can then be constructed in time

$$(\log\hat{L})\left[2^{\omega}l\left(\prod_k n_k\right)\log(md)\right]^{O(1)} + \mathrm{Time}\left(\bar{L}, D, r, \frac{\varepsilon}{l}, N_1\right) + \mathrm{Time}\,(\mathbb{P}, N_2)$$

if $(\hat{L}^2 + N_1)(md)^{2^{O(\omega)}l\prod_k n_k}$ processors are used for bit operations and $N_2(md)^{2^{O(\omega)}l\prod_k n_k}$ processors are used for calls (for any $N_1, N_2 \geqq 1$). The points $\{y^{(i)}\}_i$ will then have rational coordinates with numerator and denominator bounded in bit length by $O(\log(l) + |\log(r)| + |\log(\varepsilon)|)$.

Proof. Replace the quantifier free formula $P(y, x^{[1]}, \cdots, x^{[\omega]})$ in (1.6) with the formula

$$\bar{P}(y, x^{[1]}, \cdots, x^{[\omega]}) := P(y, x^{[1]}, \cdots, x^{[\omega]}) \wedge (\|y\|^2 \leqq r^2).$$

Let $\{h_i\}_i$ denote the set of polynomials $h_i: \mathbb{R}^l \to \mathbb{R}$ as constructed in Proposition 2.2, assuming that $\{g_i\}_i$ is replaced with $\{g_i\}_i \cup \{y \mapsto \|y\|^2 - r^2\}$, and let $\mathcal{R}\{h_i\}_i$ denote the set of polynomials $R: \mathbb{R}^{l+1} \to \mathbb{R}$ as in Proposition 2.3.

Propositions 2.2, 2.3, and 3.1 together easily imply that one can efficiently construct a set $\{y^{(i)}\}_i$ of $(md)^{2^{O(\omega)}l\prod_k n_k}$ points $y^{(i)} \in \mathbb{R}^l$ with the following property; for each connected component of SOLUTIONS (r) there exists $y^{(i)}$ within distance ε of the component.

All that remains to be accomplished is the discarding of those points $y^{(i)}$ that are not within distance ε of any connected component of SOLUTIONS (r). Determining which $y^{(i)}$ to discard is equivalent to determining which sentences

$$(\exists y \in \mathbb{R}^l)(Q_1 x^{[1]} \in \mathbb{R}^{n_1}) \cdots (Q_\omega x^{[\omega]} \in \mathbb{R}^{n_\omega})[\bar{P}(y, x^{[1]}, \cdots, x^{[\omega]}) \wedge (\|y - y^{(i)}\|^2 \leqq \varepsilon^2)]$$

are false. Relying on the algorithm of Theorem 1.1, this can be accomplished efficiently.

The operation and time bounds stated in the theorem are easy consequences of the propositions cited along with the bounds provided by Theorem 1.1. □

In closing this section we complete the proof of Proposition 1.3. Let $\{h_i\}_i$ denote the set of polynomials $h_i: \mathbb{R}^l \to \mathbb{R}$ as constructed in Proposition 2.2 (not assuming that $\{g_i\}_i$ is replaced as in the above proof). Let $\mathcal{R}\{h_i\}_i$ denote the set of polynomials $R: \mathbb{R}^{l+1} \to \mathbb{R}$ as in Proposition 2.3. Then for each connected component of SOLUTIONS, there exists $R \in \mathcal{R}\{h_i\}_i$ such that R factors linearly $R(U) = \prod_i \xi^{(i)} \cdot U$, where for some i, Aff $(\xi^{(i)})$ is in the component. If the coefficients occurring in the formula (1.6) are all integers of bit length at most L, Propositions 2.2 and 2.3 show that the coefficients of the corresponding univariate polynomials (3.1) and (3.2) obtained from $R \in \mathcal{R}\{h_i\}_i$ will be integers of bit length at most $L(md)^{2^{O(\omega)}l\prod_k n_k}$. Lemma 1.3 is now a consequence

of the observations immediately following the proof of Proposition 3.1 along with Lemma 2.9.

4. Bounds for univariate polynomial zero approximation. In this section we briefly review some known upper bounds on the operations and time required to compute approximations to the real zeros of real univariate polynomials.

Recall that P_d was defined as the set of all nonconstant real univariate polynomials of degree at most d, and $P_{L,d}$ was defined as the subset of P_d consisting of polynomials, all of whose coefficients are integers of bit length at most L.

Perhaps the best-known method for approximating real zeros of univariate polynomials is via Sturm sequences (e.g., see Henrici [14]). Given $f \in P_d$ and $0 < \varepsilon < r$, the method proceeds by bisection and computes ε-approximations to all of the real zeros $\bar{s}$ of f satisfying $|\bar{s}| \leqq r$. The number of operations required is only $d^{O(1)} \log(1+(r/\varepsilon))$; the operations can be implemented in parallel requiring time $[\log(dr/\varepsilon)]^{O(1)}$ if $d^{O(1)}$ parallel processors are used.

For readers unfamiliar with Sturm sequences, similar bounds can be obtained by invoking Theorem 1.1 to design an algorithm for approximating the real zeros of univariate polynomials. First, the algorithm of Theorem 1.1 is used to determine if the interval $[-r, r]$ contains a zero of f; of course, this is equivalent to determining if the sentence

$$(\exists s \in \mathbb{R})[(f(s)=0) \wedge (s \leqq r) \wedge (s \geqq -r)]$$

is true. If the sentence is false we terminate. Otherwise, we bisect the interval and query which of the two smaller intervals contains a zero, and so on. This approach, together with Theorem 3.2, produces the second algorithm of Theorem 1.2.

The bit complexity algorithm of Theorem 1.2 is obtained by combining Theorem 3.2 with Neff [16]. As mentioned in the introduction, Neff showed that there is an efficient parallel bit-model algorithm for approximating all zeros of a complex univariate polynomial; assuming ε is an integral power of 2, he showed that ε-approximations to all zeros can be obtained in time $[\log(L+d+|\log(\varepsilon)|)]^{O(1)}$ using $(L+d+|\log(\varepsilon)|)^{O(1)}$ processors. Taking the real part of each of these approximations, we obtain a set of real numbers containing ε-approximations to all of the real zeros of the polynomial; although some of the real numbers in the set thus obtained may not approximate any zero, Theorem 3.2 is still applicable—the algorithm developed to prove the theorem "weeds out" points which are not approximations.

In Renegar [17], an algorithm is presented that obtains approximations to all zeros (including the complex zeros) of a polynomial $f \in P_d$. The main results of that paper are presented assuming that an upper bound R on the absolute values of all of the zeros is known a priori. The operation bound presented is of the form $d^{O(1)} \log \log(3+(R/\varepsilon))$. (Specific small exponents are presented rather than relying on "$O(1)$.") The significance of the bound is its extremely low dependence on R/ε; as discussed in the introduction of the present paper, it is proven in [17] that this dependence is optimal.

Although the main results in [17] are stated in the introduction of that paper under the assumption that an upper bound R on the absolute values of all of the zeros is known a priori, the algorithm and analysis of § 8 of that paper were written to establish the following; if $f \in P_d$, $x \in \mathbb{C}$, and $r > 0$ are such that no zeros of f are contained in the region $\{y \in \mathbb{C};\ r < \|y-x\| \leqq 80d^5 r\}$, then ε-approximations to all zeros of f in $\{y;\ \|y-x\| \leqq r\}$ can be obtained with $d^{O(1)} \log\log(3+(r/\varepsilon))$ operations.

In the application of the present paper we are given $r > 0$ and wish to obtain a set of $d^{O(1)}$ points containing ε-approximations to all real zeros $\bar{s}$ of f satisfying $|\bar{s}| \leqq r$.

If we know a priori that the region $\{y\in\mathbb{C}; r<\|y\|\leqq 80d^5r\}$ contains no zeros of f, then, by the preceding paragraph, this can be accomplished quickly (i.e., quickly in terms of r and ε). If the region does contain a zero of f (as can be efficiently determined using the algorithm of Theorem 1.1), then we can design a simple bisection algorithm, that calls on the decision algorithm of Theorem 1.1, to construct $k\leqq d$ points $x_1,\cdots,x_k\in[-r,r]$ and radii $r_1,\cdots,r_k\leqq r$ with the properties that (i) all real zeros of f that are contained in $[-r,r]$ are contained in $\bigcup_j [x_j-r_j, x_j+r_j]$ and (ii) if $\bar{s}\in\mathbb{C}$ satisfies $f(\bar{s})=0$ and $\|\bar{s}-x_j\|>r_j$, then $\|\bar{s}-x_j\|\geqq 80d^5r_j$. This construction will only require $d^{O(1)}$ operations. For these smaller intervals we can rely on the algorithm of § 8 of Renegar [17] to obtain approximations to the zeros within. The observation of the preceding paragraph shows that for each of these smaller intervals, all zeros within the interval can be approximated quickly. In all, relying on the bounds of Theorem 1.1 and the preceding paragraph, ε-approximations to all zeros within the interval $[-r,r]$ can be obtained with $d^{O(1)}\log\log(3+(r/\varepsilon))$ operations. This result, combined with Theorem 3.2 yields the first algorithm in Theorem 1.2.

REFERENCES

[1] S. J. Berkowitz, *On computing the determinant in small parallel time using a small number of processors*, Inform. Process. Lett., 18 (1984), pp. 147-150.

[2] A. Borodin, J. von zur Gathen, and J. Hopcroft, *Fast parallel matrix and* GCD *computations*, Inform. Control, 52 (1982), pp. 241-256.

[3] L. Blum, M. Shub, and S. Smale, *On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines*, Bull. Amer. Math. Soc., 21 (1989), pp. 1-46.

[4] L. Blum and S. Smale, *The Godel incompleteness theorem and decidability over a ring*, in From Topology to Computation; Proceedings of the Smalefest, Springer-Verlag, New York, to appear.

[5] W. Brown and J. Traub, *On Euclid's algorithm and the theory of subresultants*, J. Assoc. Comp. Mach., 18 (1971), pp. 505-514.

[6] J. Canny, *The Complexity of Robot Motion Planning*, MIT Press, Cambridge, MA, 1988.

[7] A. L. Chistov and D. Y. Grigor'ev, *Subexponential time solving systems of algebraic equations*, I *and* II, LOMI, E-9-83, E-10-83, Leningrad, preprint.

[8] G. E. Collins, *Quantifier elimination for real closed fields by cylindrical algebraic decomposition*, Second GI Conference on Automata Theory and Formal Languages, Lecture Notes in Comput. Sci., 33 (1975), pp. 134-183, Springer-Verlag.

[9] L. Csanky, *Fast parallel matrix inversion algorithms*, SIAM J. Comput., 5 (1976), pp. 618-623.

[10] M. Coste and M. F. Roy, *Thom's lemma, the coding of real algebraic numbers and the computation of the topology of semi-algebraic sets*, J. Symbolic Comput., 5 (1988), pp. 121-129.

[11] D. Yu. Grigorev and N. N. Vorobjov, *Solving systems of polynomial inequalities in subexponential time*, J. Symbolic Comput., 5 (1988), pp. 37-64.

[12] D. Grigor'ev, *The complexity of deciding Tasrki algebra*, J. Symbolic Comput., 5 (1988), pp. 65-108.

[13] J. Heintz, M.-F. Roy, and P. Solernó, *Sur la complexité du principe de Tarski-Seidenberg*, Bull. Soc. Math. France, 188 (1990), pp. 101-126.

[14] P. Henrici, *Applied and Computational Complex Analysis*, Vol. 1, Wiley-Interscience, New York, 1974.

[15] D. Lazard, *Résolution des systèmes d'équations algébriques*, Theoret. Comput. Sci., 15 (1981), pp. 77-110.

[16] C. A. Neff, *Specified precision polynomial root isolation is in* NC, Proceedings of the 21st Annual IEEE Symposium on the Foundations of Computer Science, 1990; J. Comput. System Sci., to appear.

[17] J. Renegar, *On the worst-case arithmetic complexity of approximating zeros of polynomials*, J. Complexity, 3 (1987), pp. 90-113.

[18] ———, *On the worst-case arithmetic complexity of approximating zeros of systems of polynomials*, SIAM J. Comput., 18 (1989), pp. 350-370.

[19] ———, *On the computational complexity and geometry of the first order theory of the reals. Part* I: *introduction: preliminaries; the geometry of semi-algebraic sets; the decision problem for the existential theory of the reals*, J. Symbolic Comput., 13 (1992), pp. 255-299.

[20] J. RENEGAR, *On the computational complexity and geometry of the first order theory of the reals. Part* II: *the general decision problem*; *preliminaries for quantifier elimination*, J. Symbolic Comput., 13 (1992), pp. 301-327.

[21] ———, *On the computational complexity and geometry of the first order theory of the reals. Part* III: *quantifier elimination*, J. Symbolic Comput., 13 (1992), pp. 329-352.

[22] ———, *Recent progress on the complexity of the decision problem for the reals*, in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 6 (1991), pp. 287-308, American Mathematical Society, Providence, RI.

[23] A. TARSKI, *A Decision Method for Elementary Algebra and Geometry*, University of California Press, Berkeley, 1951.

[24] N. N. VOROBJOV JR., *Bounds of real roots of a system of algebraic equations*, Notes of Sci. Seminars of Leningrad Dept. of Math. Steklov Inst., 137 (1984), pp. 7-19.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1026-1046, December 1992
© 1992 Society for Industrial and Applied Mathematics
003

AN $O(n^2 \log n)$ ALGORITHM FOR THE HAMILTONIAN CYCLE PROBLEM ON CIRCULAR-ARC GRAPHS*

WEI-KUAN SHIH†, T. C. CHERN†, AND WEN-LIAN HSU‡

Abstract. A circular arc family F is a collection of arcs on a circle. A circular-arc graph is the intersection graph of an arc family. A Hamiltonian cycle (HC) in a graph is a cycle that passes through every vertex exactly once. This paper presents an $O(n^2 \log n)$ algorithm to determine whether a given circular-arc graph contains an HC. This algorithm is based on two subroutines for interval graphs: (i) a linear time greedy algorithm for the node disjoint path cover problem and (ii) a linear time HC algorithm. If the given graph does not contain an HC, this paper can produce a proof either through the deletion of an appropriate cutset or through the failure to obtain a specific type of HC.

Key words. graph, path, algorithm, Hamiltonian cycle

AMS(MOS) subject classifications. 68Q25, 68R05, 68R10

1. Introduction. Circular-arc graphs are rich in combinatorial structure. Various characterization and optimization problems on circular-arc graphs have been studied (some of them listed in [2]). A Hamiltonian cycle (HC) in a graph is a cycle that passes through every vertex exactly once. The problem of testing whether a graph contains a Hamiltonian cycle is NP-complete for general graphs. It remains so for special classes of graphs such as 3-connected planar graphs and bipartite graphs. In this paper, we present an $O(n^2 \log n)$ algorithm for the Hamiltonian cycle problem on circular-arc graphs.

A circular arc family F is a collection of arcs on a circle. Denote by $G=(V, E)$, a graph with a finite vertex set V and a set E of edges connecting vertices of G. The notations used in this paper can be found in standard graph theory text. A vertex is also referred to as a node. A graph G is a *circular-arc graph* if there is a circular arc family F and a one-to-one mapping of the vertices of G and the arcs in F such that two vertices in G are adjacent if and only if their corresponding arcs in F overlap. If there exists a point on the circle such that no arc in F passes through, then the corresponding graph is also called an *interval graph*. For convenience, we shall consider arcs in the family F rather than vertices in its corresponding graph G. Let n be the number of arcs in F.

Without loss of generality, assume all arc endpoints are distinct and no arc covers the entire circle. Label the n arcs arbitrarily from 1 through n. Starting with any endpoint, label all the endpoints from 1 to $2n$ according to their clockwise order. Denote an arc i that begins at endpoint p and ends at endpoint q in the clockwise direction by (p, q). Define p to be the *head* (or counterclockwise endpoint, denoted by $h(i)$) of the arc, q to be the *tail* (or clockwise endpoint, denoted by $t(i)$). An example is shown in Fig. 1.1.

The continuous part of the circle that begins with an endpoint c and ends with d in the clockwise direction is referred to as *segment* (c, d) of the circle. We use "arc" to refer to a member of F and "segment" to refer to a part of the circle between two endpoints. An arc (p, q) of F is also regarded as the segment (p, q). Arc (p, q) is

* Received by the editors June 2, 1989; accepted for publication (in revised form) September 24, 1991. This research was supported in part by the National Science Council of the Republic of China.

† Institute of Information Science, Academia Sinica, Republic of China.

‡ Department of Industrial Engineering and Management Sciences, Northwestern University, Evanston, Illinois 60208. Present address, Institute of Information Science, Academia Sinica, Republic of China.

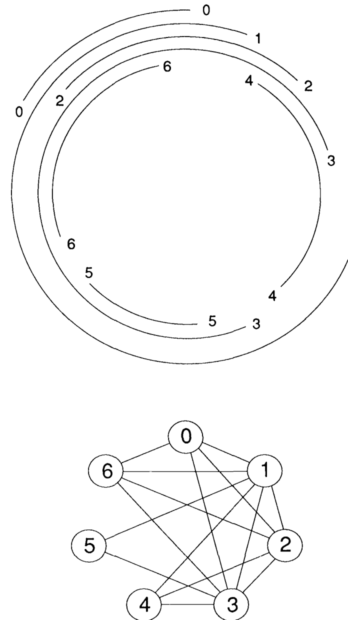


FIG. 1.1. *A circular-arc family F.*

considered *open*, namely, it contains neither p nor q. A point on the circle is said to be in arc (or segment) (p, q) if it falls within the interior of segment (p, q). An arc i of F is said to be *contained in* another arc j if every points of i is contained in arc j. When F is an interval family, we use $p < q$ to indicate that endpoint p is to the left of endpoint q.

Define a *node disjoint path cover* (NDPC) of a graph to be a collection of node disjoint paths that cover all nodes. The NDPC *problem* on a graph G is to find the minimum number (denoted by $\tau(G)$) of paths in an NDPC of G. This problem is *NP*-hard on general graphs. Linear time algorithms on interval graphs and circular-arc graphs have been proposed by Bonuccelli and Bovet [1]. Their algorithm on circular-arc graphs simply cut the circle at a point p and then apply the algorithm on the resulting interval graphs (deleting the counterclockwise portion of every arc cut by p). However, we believe there is a flaw in this approach as illustrated by the example in Fig. 1.2: if a point p is chosen in segment $(t(5), h(6))$, then the algorithm of Bonuccelli and Bovet will produce two paths $P_1 = \{6, 0, 1, 2, 3, 4\}$ and $P_2 = \{5\}$, but an optimal cover needs only one path $P = \{0, 2, 4, 1, 5, 3, 6\}$. Nevertheless, we shall adopt their NDPC algorithm on interval graphs in solving the HC problem for circular-arc graphs. It seems likely that an NDPC algorithm for circular-arc graphs can be obtained by adapting our HC algorithm.

This paper is organized as follows. Section 2 discusses the NDPC algorithm [1] for interval graphs in detail and provides an alternate proof. Based on this algorithm, we present a greedy HC algorithm for interval graphs in § 3. A preliminary HC algorithm for circular-arc graphs is presented in § 4. If that algorithm fails to identify an HC, we can either conclude that there is no HC in F or obtain a subset D of F such that $F\backslash D$ contains $|D|$ components. In the latter case, we analyze, in § 5, possible ways to connect those components of $F\backslash D$ though arcs of D in a feasible HC. Based on this analysis, a more sophisticated HC algorithm is presented in § 6.

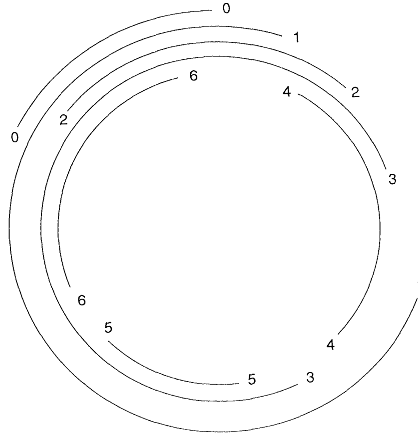


FIG. 1.2. *A counterexample for the algorithm of Bonuccelli and Bovet.*

2. An NDPC algorithm for interval graphs. Let F be a family of intervals. For each interval i, $h(i)$ is referred to as the *left* endpoint, and $t(i)$ is referred to as the *right* endpoint. A *path* P in F is denoted by a sequence of intervals $i_1 i_2 \cdots i_t$ such that i_s overlaps with i_{s+1} for $s = 1, \cdots, t-1$. When there is no confusion, P is also used to denote the set of intervals in this sequence. Our HC algorithm on circular-arc graphs makes use of the following two subroutines for interval graphs:

(a) A linear time greedy algorithm for the NDPC problem [1].

(b) A linear time HC algorithm [3], [4].

The greedy algorithm in (a) is described in Fig. 2.1. An example is shown in Fig. 2.2. The algorithm in (b) is quite similar to that in (a) and will be discussed in § 3.

GREEDY NDPC ALGORITHM FOR INTERVAL GRAPHS

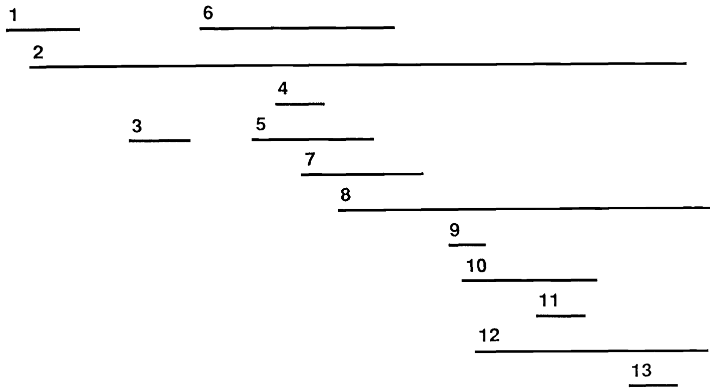
1. Sort the intervals in F into a list L according to their increasing right endpoint order. Initially, all intervals in L are unlabeled. Let the current-path-count be 1.
2. Repeat Steps 3–5 until L is empty.
3. Let the current-interval i be the first interval in L. Label i with the current-path-count and remove i from L.
4. Repeat the following until no interval in L overlaps with the current interval: label the first interval, say, j, overlapping with i by the current-path-count; remove j from L and let the current-interval i be j.
5. Increase the current-path-count by 1.
6. Output a greedy NDPC $\{P_1, P_2, \cdots, P_r\}$ (defined to be $N_{GD}(F)$), where path P_i is the ordered sequence of all intervals with label i.

FIG. 2.1. *The greedy* NDPC *algorithm for interval graphs.*

The correctness of this algorithm was first proved by Bonuccelli and Bovet [1]. Their proof is based on an inductive argument. We present an alternate proof in this section, which utilizes an important argument used throughout this paper.

THEOREM 2.1. *The size of the* NDPC $N_{GD}(F)$ *produced by the greedy* NDPC *algorithm equals* $\tau(F)$.

Our proof of this theorem will be given later. One disturbing fact about an NDPC is that intervals in different paths could overlap with each other, and a connected graph often needs to be covered by more than one path. To establish Theorem 2.1, we shall delete a subset C^* of intervals from F so that, in the remaining interval family,



$P_1 = \{1, 2, 3\}$

$P_2 = \{4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$

FIG. 2.2. *An example for the greedy* NDPC *algorithm.*

each connected component is covered by exactly one path. Each interval in C^* connects two components and is referred to as a *connector.*

Define a *cutset* C of a graph G to be a subset of nodes whose removal disconnects G. The following proposition is important in proving Theorem 2.1. Since each connected graph needs to be covered by at least one path, the proposition is easily verified by the pigeonhole principle.

PROPOSITION 2.2. *Let C be a cutset of a connected graph G. Let g be the number of connected components in $G\backslash C$. Then $\tau(F) \geqq g - |C|$.*

Define the *path-end* (respectively, *path-start*) of a path P to be the last (respectively, first) interval of P. The path-end of path P is denoted by P-end. Define the *path-gap* between two consecutive paths P_k and P_{k+1} of $N_{GD}(F)$, $k = 1, \cdots, \tau(F) - 1$ to be the segment (c, d), where $c = t(P_k\text{-end})$ and $d = h(P_{k+1}\text{-start})$ (let the last path-gap be $(t(P_{\tau(F)}\text{-end}), \infty)$). Hence, there are $\tau(F)$ path-gaps in $N_{GD}(F)$. A connected component that can be covered by one path is called a *path component.* A path component H satisfying the following condition plays an important role in our algorithm.

(2.3) *Let P be the greedy path covering H. Then $t(P\text{-end})$ is the largest among all arcs in H.*

In general, for a path P, define $I(P)$ to be the collection of all intervals in P whose tails are larger than $t(P\text{-end})$. ($t(P\text{-end})$ is the right endpoint of the path-end of P.) These are the intervals that potentially can overlap with intervals in paths after P. Define the set of *connectors* $C^*(F)$ in F recursively as follows (this set can be determined in $O(n \log n)$ time).

1. Let C denote the current set of connectors and N denote the current collection of paths. Initially, set $N = N_{GD}(F)$, $C = \emptyset$.

2. While there is a path P in N whose $I(P)$ is nonempty, perform the following. Let P_i be the first path in N with a nonempty $I(P_i)$. Include $I(P_i)$ in the current set of connectors. Delete all intervals in $I(P_i)$ from paths in the current collection N. Revise N to be the collection of resulting paths ordered according to the increasing tails of their path-ends.

Denote the final collection of paths produced in Step 2 by $N^*(F)$. Since I_p is empty for each path P in $N^*(F)$, each connected component of $F \backslash C^*(F)$ is a path component in $N^*(F)$ satisfying (2.3) (as shown in Fig. 2.3).

PROPOSITION 2.4. *Suppose* $N_{GD}(F) = \{P_1, \cdots, P_s, \cdots, P_r\}$ *(namely, F has r path-gaps). Then* $F \backslash C^*(F)$ *has* $r + |C^*(F)|$ *connected components.*

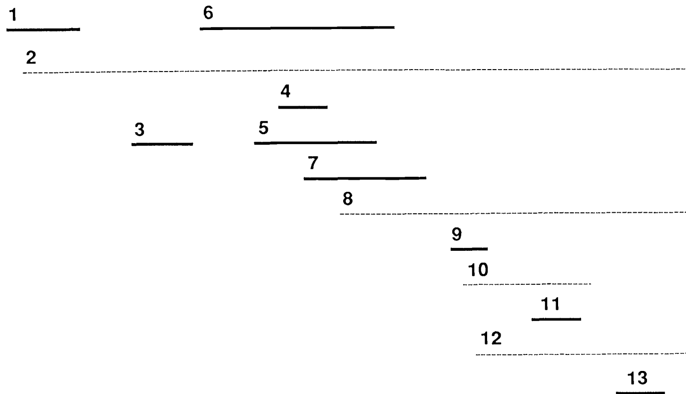
Proof. Since each component in $F \backslash C^*(F)$ can be covered by one path, it suffices to show that the number of paths in $N^*(F)$ is $r + |C^*(F)|$. The proof is by induction on $|C^*(F)|$.

If $|C^*(F)| = 0$, then each component of F is covered by exactly one path in $N_{GD}(F)$ $(= N^*(F))$, and the proposition is trivially true. Assume it holds for interval families with $|C^*(F)| \leqq k-1$, and consider a family F with $|C^*(F)| = k$. Let P_s be the first path in $N_{GD}(F)$ with a nonempty $I(P_s)$. We shall apply induction on the family $F \backslash I(P_s)$.

We shall show that $F \backslash I(P_s)$ has $r + |I(P_s)|$ path gaps. Denote $|I(P_s)|$ by r'. Arrange the intervals in F into the sequence $L_{GD}(F) = P_1 \cdots P_r$ (regard each path P_m as a sequence of intervals) ordered according to the labeling sequence of the greedy algorithm.

Claim 1. *No two intervals in* $I(P_s)$ *appear consecutively in* $L_{GD}(F)$.

Proof. Suppose there are two intervals j_1 and j_2 in $I(P_s)$ that appear consecutively in $L_{GD}(F)$ with j_1 before j_2. Since j_1 cannot be a path end, it must appear before P_s-end in $L_{GD}(F)$. But then j_2 cannot be labeled next because there exist intervals (e.g., P_s-end, labeled after j_1) overlapping j_1 whose tails are less than $t(j_2)$. □



(i) $P_1 = \{1, 2, 3\}$, $P_2 = \{4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$

Delete $\{2\}$ from P_1, $\{8, 12\}$ from P_2.

(ii) $P_1 = \{1\}$, $P_2 = \{3\}$, $P_3 = \{4, 5, 6, 7\}$,

$P_4 = \{9, 10, 11\}$, $P_5 = \{13\}$

Delete $\{10\}$ from P_4.

The resulting path components are :

$P_1 = \{1\}$, $P_2 = \{3\}$, $P_3 = \{4, 5, 6, 7\}$,

$P_4 = \{9\}$, $P_5 = \{11\}$, $P_6 = \{13\}$

$C^*(F) = \{2, 8, 10, 12\}$

FIG. 2.3. *The set of connectors of F.*

Let the intervals in $I(P_s)$ appear in the order $j_1, j_2, \cdots, j_{r'}$ in $L_{GD}(F)$, where $r' = |I(P_s)|$. By the definition of P_s, deleting $I(P_s)$ could only split P_s into subpaths, but does not change any path P_m for $m \neq s$. Hence, P_s can be rewritten as $A_1 j_1 A_2 j_2 \cdots A_{r'} j_{r'} A_{r'+1}$, where A_1 is the subsequence of P_s before j_1; A_m, for $m = 1, \cdots, r'+1$, is the subsequence in between j_m and j_{m+1}; and $A_{r'+1}$ is the subsequence after $j_{r'}$. Clearly, A_1 is nonempty. By Claim 1, A_m is also nonempty for $1 < m \leqq r'$. Finally, since P_s-end is not in $I(P_s)$, $A_{r'+1}$ is nonempty. Since P_s is a path, each A_i must be a subpath.

Thus, $L_{GD}(F) = P_1 \cdots P_s \cdots P_r = P_1 \cdots P_{s-1} A_1 j_1 A_2 j_2 \cdots A_{r'} j_{r'} A_{r'+1} P_{s+1} \cdots P_r$. Let N' be the collection of paths $\{P_1, \cdots, P_{s-1}, A_1, \cdots, A_{r'+1}, P_{s+1}, \cdots, P_r\}$.

Claim 2. *The path collection N' is the greedy* NDPC *of $F \backslash I(P_s)$, and $F \backslash I(P_s)$ has $r + r'$ path-gaps.*

Proof. To show that N' is the greedy NDPC of $F \backslash I(P_s)$, we show that $L_{GD}(F \backslash I(P_s)) = P_1 \cdots P_{s-1} A_1 A_2 \cdots A_{r'} A_{r'+1} P_{s+1} \cdots P_r$.

Clearly, the order of intervals in $P_1 \cdots P_{s-1} A_1$ in $L_{GD}(F \backslash I(P_s))$ remains unchanged. Let l_m (respectively, f_m) be the last (respectively, first) interval in A_m, $m = 1, \cdots, |I(P_s)|$. Since no interval in the subsequence $A_1 A_2 \cdots A_{r'}$ can have a tail greater than $t(P_s\text{-end})$, no interval in the subsequence $A_2 \cdots A_{r'}$ can overlap with l_1 (otherwise, j_1 would not have been selected in $L_{GD}(F)$). Furthermore, no interval in the subsequence $A_{r'+1}$ can overlap with l_1; for otherwise, they would be included in $I(P_s)$. Hence, a new path-gap is created at l_1 for $F \backslash I(P_s)$.

By the greedy algorithm on F, the first interval f_2 of A_2 must have the smallest tail among $A_2 \cdots A_{r'} A_{r'+1}$. Therefore, f_2 would be labeled immediately after l_1 by the greedy algorithm on $F \backslash I(P_s)$.

The same argument in the last paragraph can be applied to every other interval in $A_2 \cdots A_{r'} A_{r'+1}$. Furthermore, every l_i will become a new path-end in N', and no other new path-end will be created. Hence, N' is the greedy NDPC of $F \backslash I(P_s)$, and the number of path-gaps in $F \backslash P_s$ is $|N'| = |N_{GD}| + |I(P_s)| = r + r'$. □

Proof of Proposition 2.4. By the definition of "connectors," the set of connectors in $C^*(F \backslash I(P_s))$ is exactly $C^*(F) \backslash I(P_s)$. Now, apply induction on the family $F \backslash I(P_s)$. We have that $(F \backslash I(P_s)) \backslash C^*(F \backslash I(P_s))$ has $r + r' + |C^*(F \backslash I(P_s))|$ connected components. Since $(F \backslash I(P_s)) \backslash C^*(F \backslash I(P_s)) = (F \backslash I(P_s)) \backslash [C^*(F) \backslash I(P_s)] = F \backslash C^*(F)$ and $|C^*(F \backslash I(P_s))| = |C^*(F) \backslash I(P_s)| = |C^*(F)| - |I(P_s)| = |C^*(F)| - r'$, we have that $F \backslash C^*(F)$ has $r + |C^*(F)|$ connected components. □

Proof of Theorem 2.1. Combining Propositions 2.2 and 2.4, we conclude that $\tau(G) \geqq (r + |C^*(F)|) - |C^*(F)| = r$. Since the greedy NDPC algorithm produces an NDPC of size r, $\tau(G) = r$. □

The next lemma justifies the notion of "connectors" for intervals in $C^*(F)$.

LEMMA 2.5. *Deleting any subset $C_1 \subseteq C^*(F)$ from F creates $|C_1|$ additional path-gaps in $F \backslash C_1$.*

Proof. Let $\tau(F) = r$. $F \backslash C_1$ has at most $r + |C_1|$ path-gaps. To show that there are exactly $r + |C_1|$ path-gaps in $F \backslash C_1$, consider the subsequence L'_{GD} obtained by deleting those intervals in C_1 from $L_{GD}(F)$. Similar to the proof of Claim 2 in Proposition 2.4, it can be shown that L'_{GD} is the labeling sequence of the greedy NDPC algorithm on $F \backslash C_1$. □

Let $P = i_1 \cdots i_t$ be a path. The common segment $I(i_s, i_{s+1})$ of two consecutive intervals i_s and i_{s+1} in P is called their *overlapping segment*. P is called *monotone* if the tails of segments $I(i_1, i_2), I(i_2, i_3), \cdots, I(i_{t-1}, i_t)$ are nondecreasing. It is easy to verify that each path produced by the NDPC algorithm is monotone.

3. A greedy algorithm for the HC problem on interval graphs. The HC problem for interval graphs has been solved by Keil [3] in linear time, based on the linear ordering of maximal cliques. Manacher and Mankus [4] proposed another simpler HC algorithm. The approach in [4] can be considered as an adaptation of the greedy NDPC algorithm, which leads to a correctness proof similar to that of Theorem 2.1.

The main idea of the greedy HC algorithm (as shown in Fig. 3.1) can be described as follows. Let F be a collection of intervals, and let i^* be the interval with the smallest head. Let j^* be the interval with the largest tail. Starting from interval i^*, one iteratively traces out two greedy paths to the right. The path extending strategy is always to extend the shorter path first. The shorter path is the one whose path-end has the smaller tail. If these two paths can be extended to reach interval j^* after visiting all other intervals, then an HC exists. Otherwise, F can be shown to have no HC. Below, we briefly describe this greedy HC-algorithm for interval graphs. We use "current-path" to denote the shorter path and "alternate-path" to denote the longer one. An example is shown in Fig. 3.2.

GREEDY HC ALGORITHM FOR INTERVAL GRAPHS.

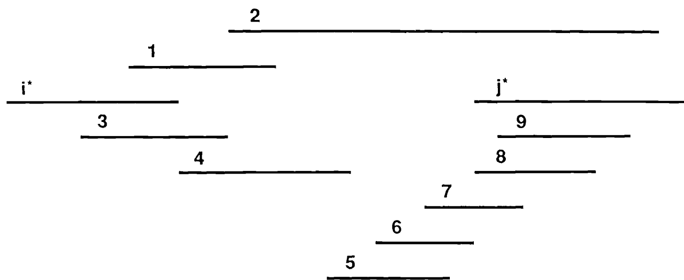
1. Sort the intervals in $F\backslash\{i^*\}$ into a list $L=\{i_1, \cdots, j^*\}$ according to their increasing "right" endpoint order. Denote the current-path by P. Denote the alternate-path by P'. If $t(i^*) < t(i_1)$, let P be $\{i^*\}$ and P' be $\{i_1\}$. Otherwise, let P be $\{i_1\}$ and P' be $\{i^*\}$. Remove i_1 from L.

2. Repeat the following until j^* is removed from L: If there exists an interval in L overlapping with P-end, attach the first such interval, say i to the end of P. Otherwise, terminate the algorithm (no HC exists). If $t(i) > t(P'\text{-end})$, let the current-path be P' and the alternate-path be P. Remove i from L.

3. (j^* has just been removed.) Apply the NDPC algorithm to L. If L can be covered by one path, attach this path to the end of P. Otherwise, terminate the algorithm (no HC exists).

4. Output two monotone paths P and P' from i^* to j^*.

FIG. 3.1. *A greedy* HC *algorithm on interval graphs.*



P = { i*, 1, 2, j* }

P' = { i*, 3, 4, 5, 6, 7, 8, 9, j* }

Note : Interval sequence { 5, 6, 7, 8, 9 } would be appended to P' in step 4.

FIG. 3.2. *An example for the greedy* HC *algorithm on interval graphs.*

The HC problem on circular-arc graphs is much harder than that on interval graphs. Our HC algorithm consists of two phases. In the first phase, we apply a simple preliminary algorithm to the given graph in § 4. This algorithm either detects an HC, shows that F does not have an HC, or identifies a subset D of arcs such that $F \backslash D$ contains $|D|$ path components. If the last case happens, we embark on the second phase. Based on the analysis in § 5, a more sophisticated algorithm is presented in § 6 to determine whether an HC actually exists.

Below, we define two special types of HCs in a circular-arc family F. Let i be an arc that passes through both endpoints of arc j but does not contain j (namely, i and j together cover the whole circle). Then there are two overlapping segments for i and j. Refer to the segment containing the head (respectively, tail) of i as the *counterclockwise* (respectively, *clockwise*) overlapping segment of i with respect to j. If two arcs overlap in only one segment, then this is called their clockwise overlapping segment. An HC in a circular-arc family is called *monotone* (as shown in Fig. 3.3) if

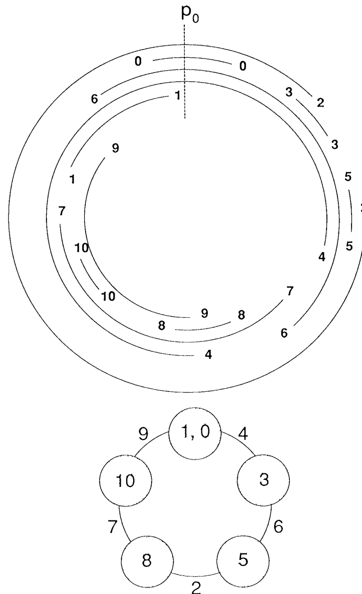


FIG. 3.3. *A monotone Hamiltonian cycle.*

(i) The clockwise endpoints of consecutive (along the HC) clockwise overlapping segments are clockwise ordered around the circle (two such consecutive endpoints are allowed to be the same);

(ii) For every three consecutive arcs i, j, and k in the HC, the two clockwise endpoints of the two consecutive clockwise overlapping segments of i with respect to j and of j with respect to k are clockwise ordered within arc j.

The second type of HCs are called *crescent*. Let p be any point on the circle. Any arc i passing through p can be divided into two segments $(h(i), p)$ and $(p, t(i))$, called

the *head portion* (denoted by $\lceil i)$) and the *tail portion* (denoted by $i\rceil)$) of i, respectively. A Hamiltonian cycle C^h on a circular-arc family is called *crescent* if there exists a point p on the circle so that arcs passing through p can be partitioned into V_1 and V_2 such that C becomes an HC of the interval graph obtained by chopping off the head portion of every arc in V_1 and the tail portion of every arc in V_2. An example is shown in Fig. 3.4. By this definition, all HCs in an interval family are crescent. In § 5, we show that if F has an HC, then there exists one which is either monotone or crescent.

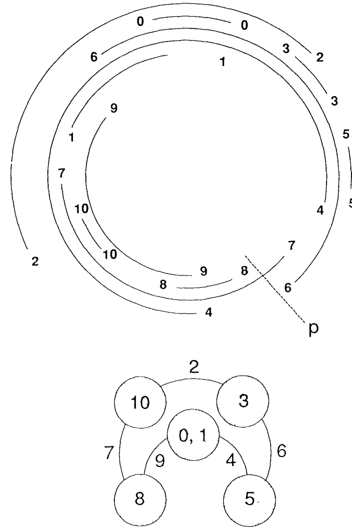


FIG. 3.4. *A crescent Hamiltonian cycle.*

4. A preliminary HC algorithm for circular-arc graphs. From now on, we assume that the given family F is not an interval family (otherwise, the HC algorithm in § 3 can be applied). A preliminary algorithm is presented in this section, which attempts to find a monotone HC in F. This algorithm may fail to identify a monotone HC even when one exists. However, in that situation, additional information will be obtained by the algorithm that enables us to embark on the more sophisticated algorithm in § 6. Specifically, at termination, the preliminary algorithm produces one of the following: (a) a monotone HC; (b) a proof that no HC exists in F; (c) a cutset D such that $F\backslash D$ contains $|D|$ connected path components each satisfying (2.3). Those components in (c) can then be used in § 6 for further analysis. The next observations can be easily proved by the pigeon hole principle and is used throughout the remainder of this paper.

LEMMA 4.1. *Suppose F contains a cutset C such that $F\backslash C$ contains more than $|C|$ connected components. Then F does not have an* HC.

The main idea of the algorithm can be described as follows. Arbitrarily pick a point p_0 on the circle. Denote by W the set of arcs passing through p_0. Every arc i in W is divided by p_0 into its *head portion* $(h(i), p_0)$ and *tail portion* $(p_0, t(i))$. For any subset W' of W, define $\lceil\mathbf{W}'$ (respectively, $\mathbf{W}'\rceil$) to be the set of head (respectively, tail) portions of arcs in W'. The algorithm attempts to find a suitable partition of W into W_1 and W_2 such that the interval family $H_w = W_1\rceil \cup \lceil W_2 \cup (F\backslash W)$ (imagine cutting the circle at p_0 and straightening all arc segments to be intervals) can be covered by one path, and the path-end belongs to $\lceil W_2$ (which winds back to p_0 and completes a monotone HC). Note that intervals in $W_1\rceil$ start from the left and of H_w and those in $\lceil W_2$ end at the right end of H_w. For convenience, we view endpoints of intervals in

H_w as going from "left to right" in a line, and use the usual "small or large" to compare their relative positions.

Initially, set $W_1 = W$ and $W_2 = \varnothing$. If the corresponding interval family H_w cannot be covered by a path with its path-end in $\lceil W_2$, then we try to extend the first path P in the greedy NDPC of H_w by moving one arc whose head portion overlaps with P-end from W_1 to W_2 (this results in one less interval on the left end of H_w, but one more on the right end). Note that intervals not in P must have a tail greater than $t(P\text{-end})$. Repeat this extension process (and keep moving arcs from W_1 to W_2) until one of the following situations happens: (1) we can successfully cover the current H_w by one path P with P-end$\in \lceil W_2$ (a monotone HC is obtained); (2) no arc in the current W_1 has its head portion overlapping with P-end (no HC exists); or (3) after moving the last arc from W_1 to W_2, we fail to extend the first path P beyond its previous path-end (a cutset D is then determined). A detailed description of the algorithm is given in Fig. 4.1.

PRELIMINARY HC ALGORITHM (for arc families that are not interval families).

1. First check that F is not an interval family. Set the iteration count, s, initially at 0. Let W_1^0 be W and W_2^0 be the empty set (and gradually, move arcs from W_1 to W_2 in Step 5).

2. Let H_s be the interval family $W_1^s\rceil \cup \lceil W_2^s \cup (F\setminus W)$ (for example, see Fig. 4.2). Apply the NDPC algorithm to H_s. Terminate the algorithm as soon as the first path (denoted P^s) is completed (label those unlabeled arcs in P^s).

3. If H_s is covered by P^s and P^s-end$\in \lceil W_2^s$, then a monotone HC is found (**TERMINATE**).

4. If no interval in $W_1^s\rceil$ whose corresponding arc overlaps with i_s, then F does not have an HC (**TERMINATE**).

5. If P^s-end was not labeled in the previous iteration consider the set $\mathbf{B}_s$ of arcs in W_1^s whose head portions contain $t(P^s\text{-end})$. Let $\mathbf{j}_s$ be the arc in B_s with the smallest tail. Let $W_1^{s+1} = W_1^s\setminus\{j_s\}$, $W_2^{s+1} = W_2^s \cup \{j_s\}$. Make j_s unlabeled. Go back to Step 2.

6. If P^s-end was labeled in the $(s-1)$-iteration (namely, the algorithm fails to extend the first path beyond P^{s-1}-end in the current iteration), consider the first path P^{s-1} in H_{s-1}. Let $C^*(P^{s-1})$ be the connector set of P^{s-1} in H_{s-1}. Let $\{Q_1, \dots, Q_t\}$ be the path components in $P^{s-1}\setminus C^*(P^{s-1})$ ordered by the increasing tails of their path-ends in H_{s-1}. If either $j_{s-1}\rceil \notin Q_1$, or $j_{s-1}\rceil \in Q_1$ but $Q_1\setminus\{j_{s-1}\rceil\}$ cannot be covered by one path, then F does not have an HC (**TERMINATE**). Otherwise, go to Step 7.

7. Determining a subset D in the case that $j_{s-1}\rceil \in Q_1$ and $Q_1\setminus\{j_{s-1}\rceil\}$ can be covered by one path. Find the connector set $C^*(Q_1\setminus\{j_{s-1}\rceil\})$. Other details will be described in the remainder of this section (**TERMINATE**).

FIG. 4.1. *A preliminary* HC *algorithm.*

We claim that, if the algorithm terminates in Step 3, then a monotone HC is found (a trivial case); if it terminates in Steps 4 or 6, then no HC exists; and if it terminates in Step 7, then a subset D of arcs can be produced such that $F\setminus D$ contains "at least" $|D|$ connected components. Note that in the last case, if $F\setminus D$ contains more than $|D|$ components, then no HC exists by Lemma 4.1. Below, we first consider the case that the algorithm terminates in Step 4.

LEMMA 4.2. *If no arc in W_1^s overlaps with i_s in Step* 4, *then F does not have an* HC.

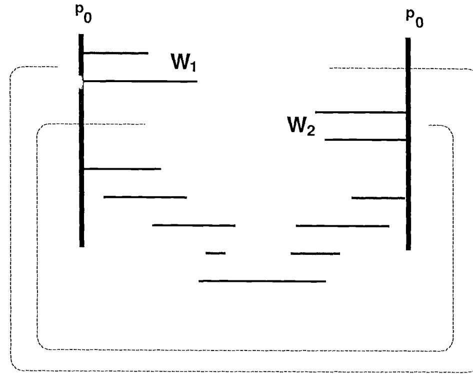


FIG. 4.2. *The interval family* H_s.

Proof. Consider the connector set $C^*(P^s)$. Let $\{PC_1, PC_2, \cdots, PC_r\}$ be the path components of $P^s \backslash C^*(P^s)$ ordered by the increasing tails of their path-ends. By Lemma 2.5, $r = |C^*(P^s)| + 1$. Denote $[F \backslash C^*(P^s)] \backslash \bigcup_{m=2}^{r} \{PC_m\}$ by PC'. Since arcs in $\bigcup_{m=2}^{r} \{PC_m\}$ must have their heads different from p_0, none of them belong to W. By the assumption of this lemma, no arc in PC' overlaps with any arc in $\bigcup_{m=2}^{r} \{PC_m\}$. Hence, the family $F \backslash C^*(P^s) = PC' \cup [\bigcup_{m=2}^{r} \{PC_m\}]$ contains at least r connected components. Since there are only $r-1$ arcs in the cutset $C^*(P^s)$, F does not have an HC by Lemma 4.1. □

LEMMA 4.3. *If the preliminary algorithm terminates in Step* 6, *then F has no* HC.

Proof. By Lemma 2.5, $t = |C^*(P^{s-1})| + 1$. Note that $j_{s-1}] = (p_{0,t}(j_{s-1}))$. Consider the following two cases.

(1) If $j_{s-1}] \notin Q_1$, then $j_{s-1}] \in C^*(P^{s-1})$. Since $t(j_{s-1})$ is the smallest tail among arcs in B_{s-1}, $B_{s-1} \subseteq C^*(P^{s-1})$. Denote $[F \backslash C^*(P^{s-1})] \backslash \bigcup_{m=2}^{t} \{Q_m\}$ by Q'. Since Q' does not contain any arc in B_{s-1}, no arc in Q' overlaps with any arc in $\bigcup_{m=2}^{t} \{Q_m\}$. Hence, $F \backslash C^*(P^{s-1}) = Q' \cup [\bigcup_{m=2}^{t} \{Q_m\}]$ contains at least t connected components. Since there are only $t-1$ arcs in the cutset $C^*(P^{s-1})$, F does not have an HC by Lemma 4.1.

(2) If $j_{s-1} \in Q_1$ but $Q_1 \backslash \{j_{s-1}]\}$ cannot be covered by one path (it requires two paths), let Q_1' be $Q_1 \backslash \{j_{s-1}]\}$. Consider the ordered path components $\{P_1, \cdots, P_{t'}\}$ in $Q_1' \backslash C^*(Q_1')$. By Lemma 2.5, $t' = |C^*(Q_1')| + 2$.

If $t(j_{s-1}) < t(P_1\text{-end})$, then adding $j_{s-1}]$ back to P_1 would result in that the tail of the first path-end for $P_1 \cup \{j_{s-1}]\}$ is no larger than $t(P_1\text{-end})$. This would imply that $Q_1 = Q_1' \cup \{j_{s-1}]\}$ requires at least two paths to cover (imagine adding intervals in $C^*(Q_1')$ one by one back to $\{P_1, \cdots, P_{t'}\}$), a contradiction. Hence, $t(j_{s-1}) > t(P_1\text{-end})$ and $B_{s-1} \subseteq C^*(F')$. Denote $C^*(F') \cup C^*(P^{s-1}) \cup \{j_{s-1}\}$ by C'. Denote $F \backslash [[\bigcup_{m=2}^{t'} \{P_m\}] \cup [\bigcup_{m=2}^{t} \{Q_m\}] \cup C']$ by P'. By the above discussion, no arc in P' overlaps with any arc in $[\bigcup_{m=2}^{t'} \{P_m\}] \cup [\bigcup_{m=2}^{t} \{Q_m\}]$.

The family $F \backslash C' = P' \cup [\bigcup_{m=2}^{t'} \{P_m\}] \cup [\bigcup_{m=2}^{t} \{Q_m\}]$ contains at least $1 + (t'-1) + (t-1) = t' + t - 1$ connected components. Since there are only $(t'-2) + (t-1) + 1 = t' + 2 - 2$ arcs in the cutset C', F does not have an HC by Lemma 4.1. □

In the remainder of this section, assume the algorithm terminates in Step 7 at the sth iteration. Recall that this happens when the algorithm fails at the sth iteration to extend the first path beyond i_{s-1}, the P^{s-1}-end, at the $(s-1)$th iteration. Furthermore, $j_{s-1}] \in Q_1$ and $Q_1 \backslash \{j_{s-1}]\}$ can still be covered by one path. Let Q_1' be the greedy path covering $Q_1 \backslash \{j_{s-1}]\}$.

LEMMA 4.4. $t(Q_1'\text{-end}) < t(Q_1\text{-end})$.

Proof. If $t(Q_1'\text{-end}) = t(Q_1\text{-end})$, then, without using $j_{s-1}]$, the greedy algorithm could still produce the first path-end for P^{s-1} at i_{s-1} (by following exactly the same sequence for P^{s-1} after Q_1'-end). But then, the head portion $[j_{s-1}$ could be used to extend the first path of H_s beyond P^{s-1}-end, and P^s-end should have been unlabeled at the $(s-1)$th iteration, contradictory to the assumption of Step 6. □

Lemma 4.4 guarantees that the connector set $C^*(Q_1')$ is nonempty. Let $\{P_1, \cdots, P_r\}$ be the path components in $Q_1' \backslash C^*(Q_1')$ ordered by the increasing tails of their path-ends in H_s.

LEMMA 4.5. $t(P_1\text{-end}) < t(j_{s-1})$.

Proof. Suppose otherwise. Then, adding $j_{s-1}]$ back to P_1 would result in the tail of the first path-end for $P_1 \cup \{j_{s-1}]\}$ being no larger than $t(P_1\text{-end})$. Hence, adding $j_{s-1}]$ back to Q_1' would result in $t(Q_1\text{-end}) \leqq t(Q_1'\text{-end})$ (imagine adding interval in $C^*(Q_1')$ one by one back to $\{P_1, \cdots, P_r\}$), contrary to Lemma 4.4. □

Since $t(j_{s-1})$ is the smallest among arcs in B_{s-1}, we have $B_{s-1} \subseteq C^*(Q_1') \cup C^*(P^{s-1}) \cup \{j_{s-1}\}$. Denote $(F \backslash [C^*(Q_1') \cup C^*(P^{s-1}) \cup \{j_{s-1}\}]) \backslash ([\bigcup_{m=2}^{r} \{P_m\}] \cup [\bigcup_{m=2}^{t} \{Q_m\}])$ by P'. Then, no arc in P' overlaps with any one in $[\bigcup_{m=2}^{r} \{P_m\}] \cup [\bigcup_{m=2}^{t} \{Q_m\}]$. Let $\{R_1, \cdots, R_k\}$ be the path components in $P' \backslash C^*(P')$ ordered by the increasing tails of their path-ends in interval family P'. Let $D = C^*(P') \cup C^*(Q_1') \cup C^*(P^{s-1}) \cup \{j_{s-1}\}$. A pictorial description of the set D is given in Fig. 4.3.

THEOREM 4.6. *$F \backslash D$ has at least $|D|$ path components, each satisfying* (2.3).

Proof. By Lemma 2.5, $k \geqq 1 + |C^*(P')|$. By the discussion above, $F \backslash D$ is composed of the following path components $[\bigcup_{m=1}^{k} \{R_m\}] \cup [\bigcup_{m=2}^{r} \{P_m\}] \cup [\bigcup_{m=2}^{t} \{Q_m\}]$, the number of which equals $k + 1(r-1)^*(t-1) \geqq (1 + |C^*(P')|) + |C^*(Q_1')| + |C^*(P^{s-1})| = |D|$. □

Clearly, if $F \backslash D$ has more than $|D|$ components, then by Lemma 4.1, F has no HC. Hence, assume $F \backslash D$ has exactly $|D|$ components. Further analysis is necessary in order to determine whether an HC actually exists in F. We discuss this in §§ 5 and 6.

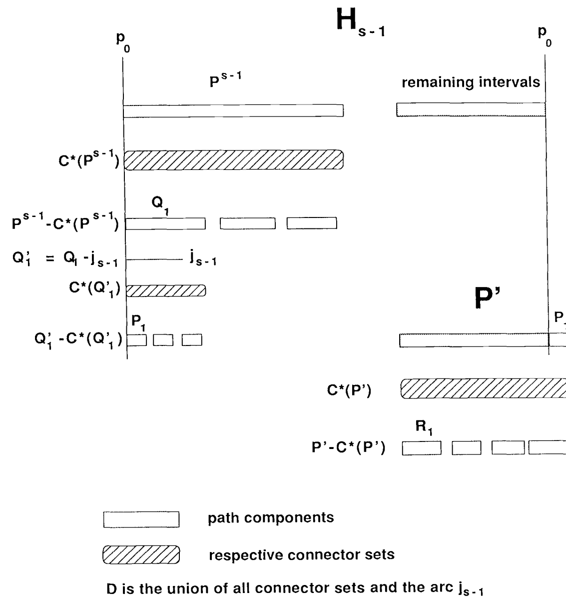


FIG. 4.3. *The set D and the path components in $F \backslash D$.*

5. Analysis of an HC in circular-arc graphs. Throughout this section, assume there is a Hamiltonian cycle C^h in F and a cutset D such that $F\backslash D$ has $|D|$ path components, each satisfying (2.3). We shall study how an HC in F traverses these $|D|$ components through arcs in D. Our analysis will establish that if F contains an HC, then there must exist either a monotone HC or a crescent HC in F. Thus, the search for an HC in § 6 can be restricted to these two types.

Our approach is to modify C^h gradually to become either monotone or crescent. Let $F' = F\backslash D$. Order the path components of F' into a clockwise circular sequence $S = C_1 C_2 \cdots C_{|D|}$. For each component C_w, let $h(C_w)$ be the most counterclockwise head of arcs in C_w and $t(C_w)$ be the most clockwise tail in C_w. Denote by (C_r, C_s) the set of all components in a "clockwise traversal" of S from (and including) C_r to C_s. (C_r, C_s) is said to be *covered by an arc i* if every point of the segment $(t(C_r), h(C_s))$ is contained in i. An example illustrating these definitions is given in Fig. 5.1. Note that in the cycle C^h each arc of D connects a unique pair of components of F', and every component is connected to others through exactly two arcs of D in C^h. We shall refer to the arcs of D as the connectors of C^h.

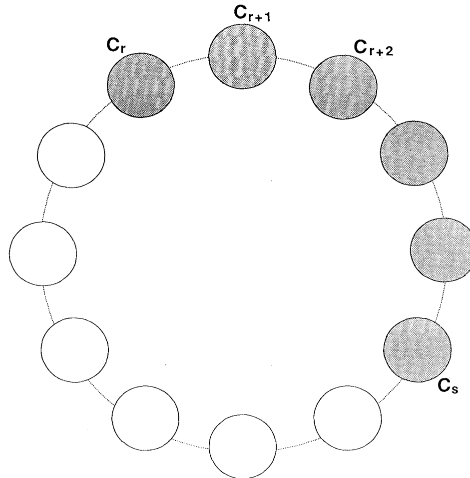


FIG. 5.1. *The set of components* (C_r, C_s).

Now, consider consecutive pairs of components in S. A consecutive pair $\{C_w, C_{w+1}\}$ is called a *type* I *pair* for C^h if there exists an arc i in D connecting two components C_r, C_s in C^h, such that i covers (C_r, C_s), and $(C_w, C_{w+1}) \subseteq (C_r, C_s)$. A consecutive pair is called *type* II for C^h if it is not type I. Our main theorem in this section is the following.

LEMMA 5.1. *F has either a monotone* HC *or a crescent* HC. (*Note that F is assumed to have an* HC *in this section.*)

Proof. In this proof, we adopt several techniques to transform an existing HC of F into either a monotone HC or a crescent HC if F has an HC. We first try to transform the HC in F into a crescent HC.

Define the *essential portion* of a connector in a given HC to be the portion of this connector from which we cannot remove any small segment without disconnecting the original HC. For a given HC, if we can find a cut point (denoted by q) in the circle such that the no essential portion of any connector crosses q, then we can transform this HC into a crescent HC as follows: cut each connector that crosses q into two arcs

and throw away the one which doesn't contain the essential portion. The remaining arcs will form an interval family, say F_q, and the original HC is an HC of F_q. A crescent HC of F_q can be found by the algorithm in § 3, which becomes a crescent HC of the original circular arc family.

In the remainder of this proof, we assume that no such cut point exists. We show that the arcs in D can be used to connect consecutive pairs of components of F' in forming a monotone HC. We apply the following four steps to modify the connecting relationships among the components of F'.

Pick C_1 as the starting component. Pick an arc d_1 in D that connects C_1 with another component, say C_w. Either (C_1, C_w) or (C_w, C_1) is covered by d_1. Without loss of generality, assume (C_1, C_w) is covered by d_1. Initially, label only C_1, C_w and leave all other components unlabeled. We shall use the following notations throughout the modification algorithm:

Current path P—the set of labeled components (viewed as *vertices*) and labeled arcs (viewed as *edges* connecting those vertices), clockwise ordered between the two end components.

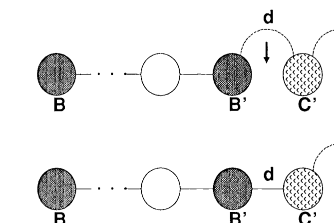
Current component C—initially, C_w. (Note that there exists an arc in P connecting C with some other component in P.)

Current arc d—the other arc of C^h which connects the current component C to an unlabeled component, say C_k.

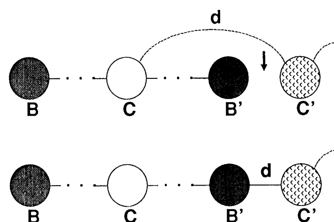
Next component C′—C_k.

Denote the two end components of the current path P by B and B' (as shown in Fig. 5.2), which are constantly updated. Initially, $B = C_1$, $B' = C_w$, and d_1 covers (B, B'). Let d be the arc in C^h that connects C_w to the next unlabeled component C'. While (B', B) is not covered by the current arc d, perform one of the following two steps.

Step 1. C' is not in (B, B') as shown in Fig. 5.2. Without loss of generality, assume B' is contained in (C, C'). If $B' \neq C$, let d connect B' and C' (otherwise, d still connects C and C'). Make C' the new boundary component B'. Label d and C'.



(i) The operations in step 1 for C = B'



(ii) The operations in step 1 for C ≠ B'

FIG. 5.2. *The operations in Step* 1.

Step 2. C' is in (B, B') as shown in Fig. 5.3. Let C_s, C_m be the two consecutively labeled components in P such that (C_s, C_m) contains C' and C_s, C_m are connected by arc d' in C^h. Without loss of generality, assume C_m is contained in (C_s, C). Let d connect C' and C_m, d' connect C_s and C'. Label d and C'.

At the end of each step, we have a path P consisting of all labeled components ordered clockwise in which each labeled arc i connects two consecutive labeled components in P. Let the current arc be the other arc of C^h, which connects C' to an unlabeled component, say C_k. Change the current component to be C'. Change the next component to be C_k.

This procedure iterates until (B', B) is covered by the current arc d. Note that the latter condition must eventually be satisfied. Otherwise, since all components will be included in P, we would end up with P being a path with $B = C_j$ and $B' = C_{j-1}$ for some j. But then, (C_j, C_{j-1}) would not be covered by the essential portion of any connector, a contradiction. Hence, as soon as (B', B) is covered by the current arc d, we adopt the following.

Step 3. (B', B) is contained in (C, C') as shown in Fig. 5.4. Let d connect B and B'. Label d. Thus, the labeled components (viewed as vertices) and labeled arcs (viewed as edges connecting vertices) form a cycle, called the *current cycle* Q. Label C'. Let the current arc be the other arc of C^h, which connects C' to an unlabeled component, say C_k. Let the current component be C'. Let the next component be C_k.

We now have a labeled cycle Q and a labeled current component outside Q. While the next component is not C_1, perform the following step.

Step 4. Let C_s, C_m be the two consecutively labeled components in Q such that (C_s, C_m) contains C as shown in Fig. 5.5. Let d' be the labeled arc that connects C_s and C_m in C^h. Without loss of generality, assume C_s is contained in (C', C). Now, form a new HC by connecting C, C_m with d', and connecting C_s, C with d. Label d and C'. Place C into cycle Q. Let the current arc by the other arc of C^h that connects C' to an unlabeled component, say C_k. Let the current component be C'. Let the next component be C_k.

At the end of Step 4, the next component becomes the starting component C_1; all components are labeled; all arcs are labeled, each connecting two consecutive components of S (see Fig. 5.6).

Now, we have created an HC in which each arc in D is used to connect two consecutive components of S. Next, construct a monotone HC by applying the NDPC algorithm within each path component of F' and using arcs in D as connectors connecting consecutive path components. A technical detail should be noted here. If

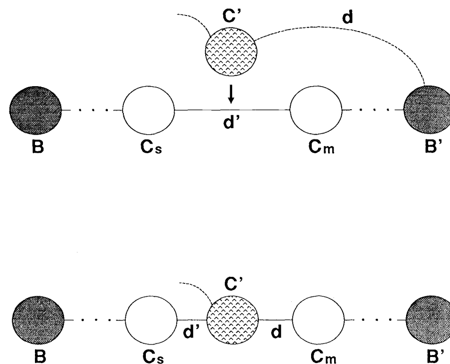


FIG. 5.3. *The operations in Step* 2.

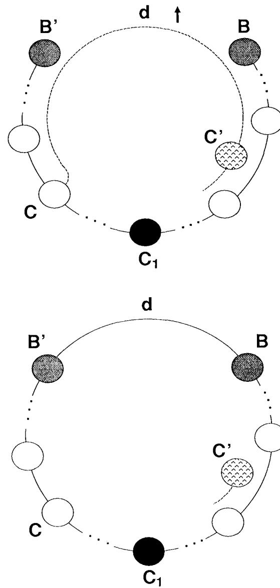


FIG. 5.4. *The operations in Step* 3.

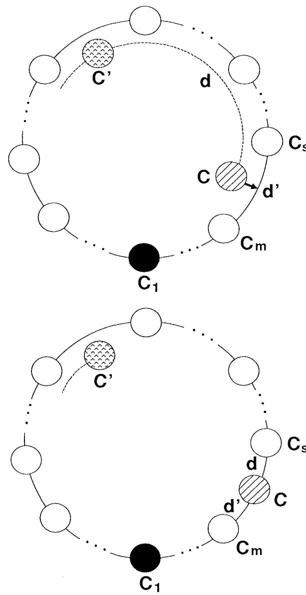


FIG. 5.5. *The operations in Step* 4.

a connector d connecting $\{C_w, C_{w+1}\}$ overlaps with the arc with the most counterclockwise tail in C_{w+1} (denoted $i(C_{w+1})$), then the NDPC algorithm on C_{w+1} would yield a path satisfying (2.3) by assumption. On the other hand, if a connector d connecting $\{C_w, C_{w+1}\}$ does not overlap with $i(C_{w+1})$, then it must be that

(a) C_{w+1} can still be covered by one path, say P_{w+1}, by applying the NDPC algorithm with the arc whose head is $h(C_{w+1})$:

(b) The connector d' connecting $\{C_{w+1}, C_{w+2}\}$ overlaps with P_{w+1}-end.

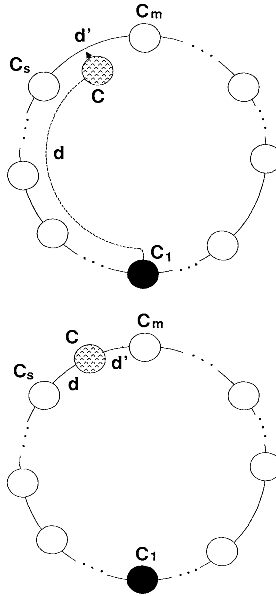


FIG. 5.6. *The operations in the final step.*

If either of these is not satisfied, then the original assumption that there is an HC is not valid, since our modification never creates such a connector d that violates one of (a) or (b). □

LEMMA 5.2. *Let $\{C_s, C_{s+1}\}$ be a dividing pair for a crescent Hamiltonian cycle C^h. If there exists at least an arc i in D which contains $(h(i(C_s)), h(C_{s+1}))$, then F has a monotone* HC.

Proof. Let $\{C_r, C_m\}$ be the two path components that i connects in C^h such that $(t(C_r), h(C_m))$ is covered by i. Then $(C_s, C_{s+1}) \not\subseteq (C_r, C_m)$. Let i' be the other arc of D that connects C_r to a component C' different from C_m in C^h. We now use the terminology in § 5. Starting with the current path P whose two end components are C_r and C', we can iteratively apply the operations in Steps 1 or 2 of § 5. Each such operation will add one more new component to the current path P. Since no arc in $D\backslash\{i\}$ can cover (C_s, C_{s+1}), we should eventually obtain a component path P^* that contains all components in S and has the two end components C_{s+1} and C_s after using all arcs in $D\backslash\{i\}$. It is easy to verify that, following the connection in P^*, we can construct a monotone path, say P', starting with $i(C_{s+1})$, traversing through all components and all arcs of $F\backslash\{i\}$ and ending with some arc, say j, in C_s. Since i contains $(h(i(C_s)), h(i(C_{s+1})))$, it must overlap with both j and $i(C_{s+1})$. Hence, arc i and path P' together form a monotone HC. □

The following theorem is an obvious consequence of Lemmas 5.1 and 5.2. This theorem is not used in § 6 (instead, Lemma 5.2 is used). We present it here for purposes of information only.

THEOREM 5.3. *If there is no type* II *consecutive pair for C^h, then F has a monotone* HC.

6. The main HC algorithm. Assume the given family F contains a subset D of arcs such that $F\backslash D$ has $|D|$ path components, each satisfying (2.3). We shall decide whether F contains an HC by searching for either a monotone HC or a crescent HC.

Our algorithm searches for a monotone HC first. If it fails to find one, then we search for a crescent HC.

First, consider the test for a monotone HC. Since there are $|D|$ arcs that are used to connect $|D|$ consecutive pairs of components in a monotone HC, one might be tempted to apply a bipartite matching algorithm in this situation. However, such an algorithm is usually not applicable because there are different ways to "connect" two consecutive pairs, and whether certain "connection pattern" forms an HC depends very much on the specific arc overlapping relationships. An example is illustrated in Fig. 6.1. The idea of our algorithm is to further restrict the connecting patterns until a useful property (condition (6.2)) of the HC emerges.

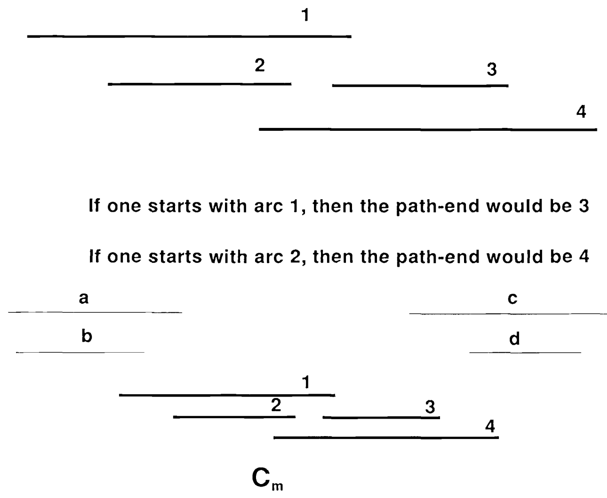


FIG. 6.1. *Different starting arcs resulting in different path-ends.*

Pick an arbitrary path component, say C_1. Let X be the set of arcs in D passing through $t(C_1)$ (not including any arc in C_1). If there exists a monotone HC C^h in F, then there is partition of X into X_h and X_t such that only the head (or tail) portion of arcs in X_h (or X_t) is used to connect two path components in C^h. Note that each arc of D must connect two consecutive path components in C^h.

Since each path component C_w in $F\backslash D$ satisfies (2.3), if we apply the greedy algorithm on C_w starting with $i(C_w)$, the path-end should always be the arc with tail $t(C_w)$. Note that $h(i(C_w))$ is not necessarily $h(C_w)$. Now, the circle is divided into $|D|$ segments by $h(C_1), \cdots, h(C_{|D|})$. The following theorem further describes certain desirable property of a monotone HC that we search for.

THEOREM 6.1. *Let C^h be a monotone* HC *with the partition X_h and X_t at $t(C_1)$. Then there exists a monotone* HC, *say C^{h^*}, satisfying*

(6.2) *There exists a pair $\{C_s, C_{s+1}\}$ connected by an arc in X_h such that in C^{h^*}, arcs in X_h (respectively, X_t) only connect those pairs $\{C_m, C_{m+1}\}$ with $m \geqq s$ (respectively, $m < s$), and no arc in $D\backslash X_h$ that connects a pair $\{C_w, C_{w+1}\}$ with $w > s$ can contain $(h(i(C_s)), h(i(C_{s+1})))$.*

Proof. Among all monotone HCs in F, let C^{h^*} be the one for which the smallest s^* such that $\{C_{s^*}, C_{s^*+1}\}$ is connected by an arc i_h of X_h in C^{h^*} is as large as possible. We claim that C^{h^*} satisfies (6.2).

Suppose there are violations to (6.2). Then there are two ways that a violation can occur, and we show that each leads to a contradiction.

(1) There is an arc i_t of X_t in C^{h^*} whose tail portion connects a pair $\{C_m, C_{m+1}\}$ in C^{h^*} with $m > s^*$. Then, by letting i_h connect $\{C_m, C_{m+1}\}$ and i_t connect $\{C_{s^*}, C_{s^*+1}\}$ (as shown in Fig. 6.2(i)), we obtain another HC $C^{h'}$ such that arcs of X_h in $C^{h'}$ only connect those pairs $\{C_m, C_{m+1}\}$ with $m \geqq s^* + 1$, a contradiction.

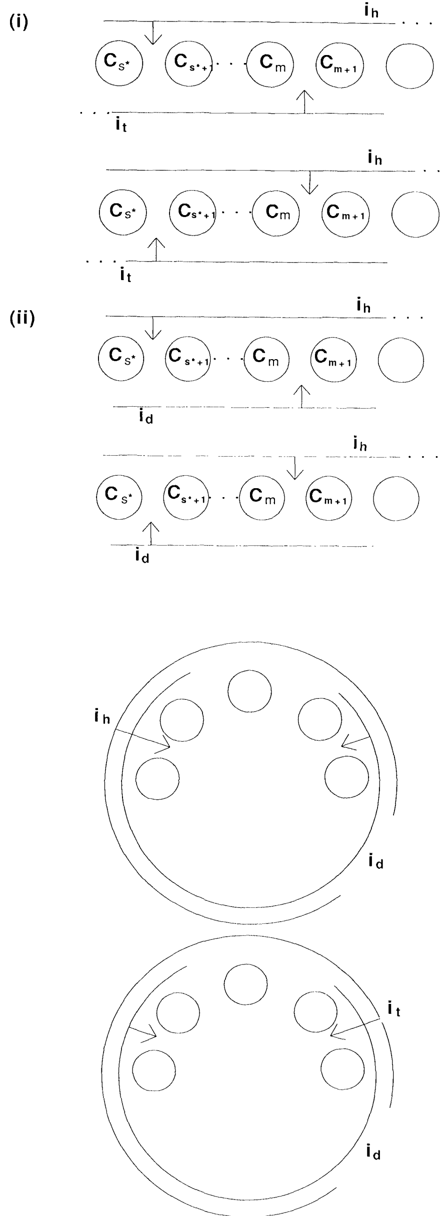


FIG. 6.2. *Change of connecting patterns.*

(2) There is an arc i_d in $D\backslash X$ that connects a pair $\{C_m, C_{m+1}\}$ in C with $m > s^*$ and i_d contains $(h(C_s), h(C_{s+1}))$. Then, by letting i_h connect $\{C_m, C_{m+1}\}$ and i_d connect $\{C_s, C_{s+1}\}$ (in Fig. 6.2(ii) and (iii)), we get a similar contradiction as in (1). □

Note that the HC C^{h^*} is constructed relative to a given HC C^h of F. The following lemma suggests an algorithm for constructing a monotone HC.

LEMMA 6.2. *Let C^h be a monotone* HC *with the partition X_h and X_t at $t(C_1)$. Let $\{C_s, C_{s+1}\}$ be a pair satisfying* (6.2) *for C^h. Let j^* be the arc in C_s with $t(j^*) = t(C_s)$. Let Y be the set of arcs passing through $t(C_s)$. Let Y_1 be the set of arcs in $Y \cap X$ not containing $(t(C_1), t(C_s))$ and Y_2 be $Y \backslash Y_1$. Let $Y_1]$ (respectively, $\lceil Y_2$) be the tail (respectively, head) portions of arcs in Y_1 with respect to $t(C_s)$. Let H_F be the interval family $Y_1] \cup \lceil Y_2 \cup (F \backslash Y)$. Then, H_F can be covered by a monotone path ending with j^*.*

Proof. We shall follow the argument in the above proof. Since each interval in D still connects the pair of path components that its corresponding arc connects in C^h, C^h reduces to a monotone path of H_F ending with j^*. □

Conversely, it is easy to see that a monotone path of H_F ending with j^* gives rise to a monotone HC in F. Therefore, a monotone HC of F can be constructed by the greedy algorithm once the pair $\{C_s, C_{s+1}\}$ satisfying (6.2) is identified. Our algorithm for finding a monotone HC in F simply checks each consecutive pair $\{C_w, C_{w+1}\}$ to see if the corresponding transformed interval family with respect to $t(C_w)$ has the desired monotone path.

Next, consider the case that F does not have a monotone HC and the above test fails. We shall embark on our second text for a crescent HC. The following analysis shows that it suffices to search for crescent HCs satisfying very special properties. Suppose F contains a crescent Hamiltonian cycle C^h. By definition, there exists a point p on the circle so that arcs passing through p can be partitioned into V_1 and V_2 such that C^h becomes a crescent HC of the interval family $V_1] \cup \lceil V_2 \cup [F \backslash (V_1 \cup V_2)]$. If p is covered by an arc in some path component C_s of $F \backslash D$, then it is easy to see that there should be at least four arcs in D that connect C_s to other path components in the cycle C^h, which is impossible (since in an HC every component is connected by exactly two arcs in D to other components). Hence p must be in an open segment between two consecutive components, say $\{C_s, C_{s+1}\}$, which is defined to be the *dividing pair for C^h*.

If F does not have a monotone HC, then by Lemma 5.2, the test for a crescent HC reduces to test if F has a dividing pair $\{C_s, C_{s+1}\}$ such that $(h(i(C_s)), h(i(C_{s+1})))$ is not covered by any arc in D. This can be easily done as follows. Take each pair $\{C_w, C_{w+1}\}$ whose $(h(i(C_w)), h(i(C_{w+1})))$ is not covered by any arc in D. Using $h(i(C_{w+1}))$ as the origin, create an interval graph H_w by chopping off the head portion of every arc $j = (a_j, b_j)$ of D whose a_j is in $(h(i(C_w)), h(i(C_{w+1})))$ and the tail portion (at $h(i(C_w))$) of every arc j of D whose b_j is in $(h(i(C_w)), h(i(C_{w+1})))$. Then test if the resulting interval graph H_w has a crescent HC.

7. Complexity analysis. Our HC algorithm consists of three subalgorithms. In the first step, we apply the preliminary HC algorithm in § 4 whose run-time is $O(n^2 \log n)$. If the preliminary HC algorithm fails to determine if there exists an HC, some components and a set of connectors are generated. We then use the algorithm in the beginning of § 6 to try to find a monotone HC. This second algorithm has at most n iterations, and each iteration takes at most $O(n \log n)$ time. Hence, the run time of the second algorithm is $O(n^2 \log n)$. If this algorithm fails to find a monotone HC, we then use the algorithm at the end of § 6 to find a crescent HC. The number of iterations in the third algorithm is at most n, and the run time for each iteration is

$O(n \log n)$. Hence, the run time of the third algorithm is $O(n^2 \log n)$. Therefore, the time complexity of our HC algorithm is $O(n^2 \log n)$.

REFERENCES

[1] M. A. BONUCCELLI AND D. P. BOVET, *Minimum node disjoint covering for circular-arc graphs*, Inform. Process. Lett., 8 (1979), pp. 159–161.

[2] M. C. GOLUMBIC, *Algorithmic Graph Theory and Perfect Graphs*, Academic Press, New York, 1980.

[3] J. M. KEIL, *Finding Hamiltonian circuits in interval graphs*, Inform. Process. Lett., 20 (1985), pp. 201–206.

[4] G. K. MANACHER AND T. A. MANKUS, *An optimum $\theta(n \log n)$ algorithm for finding a canonical Hamiltonian path and a Hamiltonian circuit in a set of intervals*, Inform. Process. Lett., 35 (1990), pp. 205–212.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1047–1069, December 1992
© 1992 Society for Industrial and Applied Mathematics
004

FULLY DYNAMIC ALGORITHMS FOR 2-EDGE CONNECTIVITY*

ZVI GALIL† AND GIUSEPPE F. ITALIANO‡

Abstract. This paper studies the problem of maintaining the 2-edge-connected components of a graph undergoing repeated dynamic modifications, such as edge insertions and edge deletions. It is shown how to test at any time whether two vertices belong to the same 2-edge-connected component, and how to insert and delete an edge in $O(m^{2/3})$ time in the worst case, where m is the current number of edges in the graph. This answers a question posed by Westbrook and Tarjan [Tech. Report CS-TR-229-89, Dept. of Computer Science, Princeton University, Princeton, NJ, August 1989; *Algorithmica*, to appear].

For planar graphs, the paper presents algorithms that support all these operations in $O(\sqrt{n \log \log n})$ worst-case time each, where n is the total number of vertices in the graph.

Key words. analysis of algorithms, dynamic data structures, edge connectivity, vertex connectivity

AMS(MOS) subject classifications. 68P05, 68Q20, 68R10

1. Introduction. An undirected graph $G = (V, E)$ is said to be *biconnected* if there are at least two vertex disjoint paths from each vertex to every other vertex. It is well known [2] that a graph G is not biconnected if and only if there is a vertex whose removal disconnects G: such a vertex is called an *articulation point*. Given a graph $G = (V, E)$, let $E_1, E_2, \ldots, E_h$ be the partition of E into equivalence classes such that two edges e' and e'' are in the same equivalence class if and only if either (i) $e' = e''$ or (ii) there is a simple cycle (i.e., a cycle with no repetition of edges and vertices) of G containing both e' and e''. For $1 \leq i \leq h$, let V_i be the set of vertices that are endpoints of the edges in E_i. The subgraphs $G_i = (V_i, E_i)$ are called the *biconnected components* (or *blocks*) of G. An articulation point appears in more than one biconnected component. The set of edges of every graph can be partitioned into biconnected components in a unique way, and every edge belongs exactly to one biconnected component. An edge contained in no cycle is in a biconnected component by itself and is referred to as a *bridge*. A graph with no bridges is called *2-edge-connected* (or *bridge-connected*). If the graph has bridges, then the removal of a bridge disconnects the graph. The *2-edge-connected components* (or *bridge-connected components* or *bridge-blocks*) of a graph G are the components of G obtained after the removal of all the bridges. Two edges e' and e'' are in the same bridge-connected component if and only if either (i) $e' = e''$ or (ii) there is a (not necessarily simple) cycle (i.e., a cycle with no repetition of edges but with possible repetition of vertices) of G containing both e' and e''. Figure 1 shows a graph together with its biconnected and bridge-connected components.

The problem of computing the bridge-connected components of undirected graphs arise naturally in many applications and have been extensively studied. Tarjan [47] gives optimal linear-time sequential algorithms. Awerbuch and Shiloach [7] and Tarjan and

*Received by the editors March 8, 1991; accepted for publication (in revised form) August 12, 1991. This work was partially supported by National Science Foundation grants CCR-8814977, CCR-9014605, by the ESPRIT II Basic Research Actions Program of the EC under contract 3075 (Project ALCOM), and by the Italian MURST Project "Algoritmi e Strutture di Calcolo." Portions of this paper appear in the Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, 1991.

†Department of Computer Science, Columbia University, 450 Computer Science Building, New York, New York 10027, and Tel-Aviv University, Tel-Aviv, Israel.

‡Department of Computer Science, Columbia University, New York, New York 10027, Università di Roma, Rome, Italy, and IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.
This author's work was partially supported by an IBM Graduate Fellowship. Part of this work was done while visiting the IBM T. J. Watson Research Center and Universität des Saarlandes, Saarbrücken, Germany.

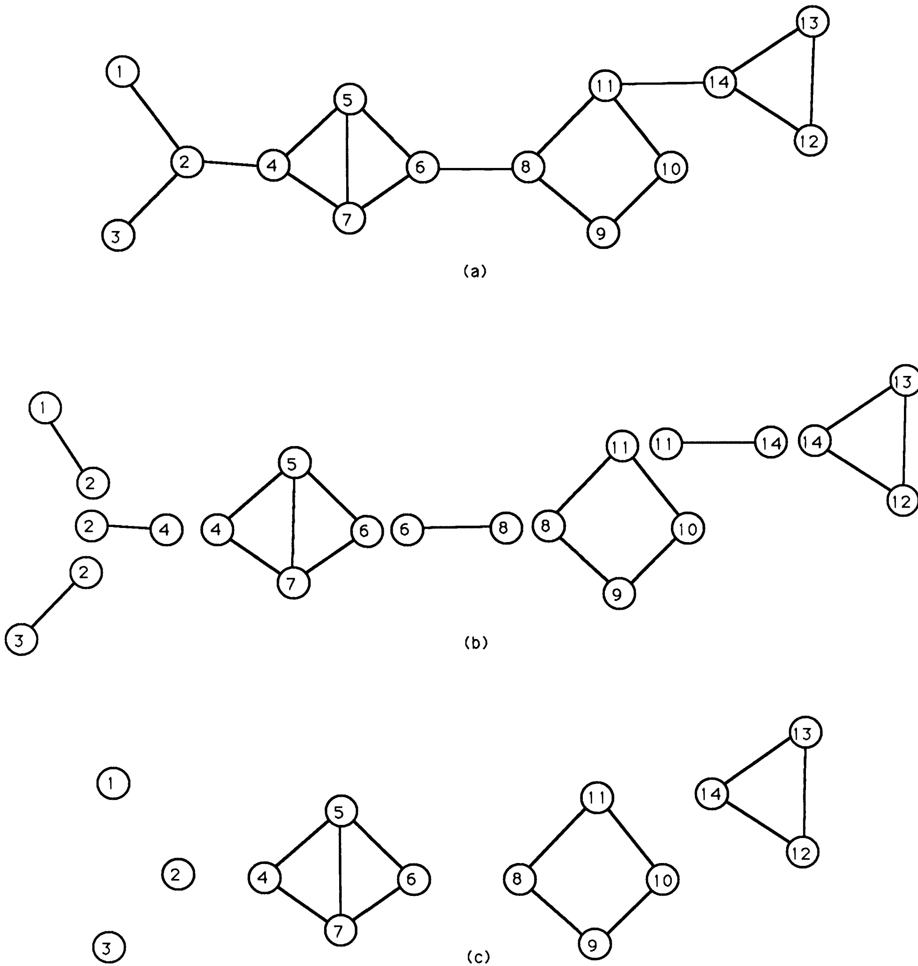


FIG. 1. (a) *A graph* $G = (V, E)$; (b) *the biconnected components of* G; (c) *the bridge-connected components of* G.

Vishkin [50] give logarithmic-time parallel algorithms. Recently, Westbrook and Tarjan [52] considered the problem of maintaining biconnected components and bridge-connected components undergoing any sequence of edge and vertex insertions. They presented algorithms that run in a total of $O(q\alpha(q, n))$ time, where q is the total number of operations, n the number of vertices, and α the inverse Ackermann's function. This bound is optimal for pointer algorithms [38], [48] as well as in the cell-probe model of computation [54].

In the last decade there has been a growing interest in dynamic problems on graphs. In these problems one would like to answer queries on graphs that are undergoing a sequence of updates, for instance, insertions and deletions of edges and vertices. A problem is often said to be *fully dynamic* if the update operations include both insertions and deletions of edges. On the other hand, a problem is called *partially dynamic* if only one type of update, i.e., either insertion or deletion, is allowed. The goal of a (either fully or partially) dynamic graph algorithm is to update efficiently the solution of a problem after dynamic changes, rather than having to recompute it from scratch each time. Given their powerful versatility, it is not surprising that dynamic algorithms and dynamic data structures are often more difficult to design than their static counterparts.

In the realm of graph problems, several dynamic data structures have been proposed to support insertions and deletions of edges and/or vertices in a graph, in addition to certain types of queries. In particular, much attention has been devoted to (among other things) the dynamic maintenance of connected components [19], [20], [39], [42], [51], [52]; 2- and 3-connectivity [15], [51], [52]; transitive closure [6], [28], [29], [30], [35], [46], [55]; planarity [14], [15], [45]; shortest paths [5], [13], [18], [36], [40], [44], [55]; and minimum spanning trees [12], [17], [20], [44]. Dynamic algorithms for graph problems are both of theoretical and of practical interest in several application areas, including communication networks, computer-aided design [3]; distributed computation, database systems [1], [32], [53]; logic programming [4], [9]; incremental data flow analysis [10], [11]; high level languages for incremental computations [56]; incremental generation of parsers [27]; and programming environments [3], [26], [34].

Fully dynamic graph problems are usually much more difficult to solve than the corresponding partially dynamic problems. For example, the partially dynamic maintenance of the connected components of an undirected graph can be done in $O(\alpha(q,n))$ amortized time per insertion using set-union data structures [49], where α is a very slowly growing function (a functional inverse of Ackermann's function), q is the total number of operations, and n is the total number of vertices in the graph. Similarly, the partially dynamic maintenance of a minimum spanning tree requires $O(\log m)$[1] time per insertion [20] where m is the current number of edges in the graph, by using the dynamic trees of Sleator and Tarjan [42], [43]. However, in both cases no better bound than $O(\sqrt{m})$ is known for the corresponding fully dynamic problems [20]. Moreover, despite intensive research on dynamic graphs, there are very few graph-theoretic problems for which a fully dynamic nontrivial algorithm is known. These are mostly problems on very restricted classes of graphs such as trees [42], [43]; series parallel graphs [13], [31]; st-planar graphs and spherical st-planar graphs with fixed embedding [45], [46]; and planar graphs with fixed embedding [17]. The only fully dynamic technique known to date for general graphs is the algorithm by Frederickson for maintaining minimum spanning trees and connected components of undirected graphs [20]. Although considerable effort has been spent on many other fundamental graph problems, a fully dynamic solution of these problems has remained elusive.

In this paper we study the problem of maintaining the 2-edge-connected components of a graph under an intermixed sequence of the following operations, starting from the null graph (i.e., a graph with n vertices and no edges).

Same2EdgeBlock(u,v): Return *true* if vertices u and v are in the same 2-edge-connected component. Return *false* otherwise.

InsertEdge(x,y): Insert a new edge between the two vertices x and y.

DeleteEdge(x,y): Delete the edge between the two vertices x and y.

Motivations for studying this problem arise from the design and management of reliable networks.

As mentioned in [52], in such a fully dynamic setting there has been no known solution better than a repeated application of an off-line algorithm, which yields a single operation worst-case time complexity of $O(m)$, where m is the current number of edges in the graph. Reif [39] introduced the notion of *complete dynamic problems*, as a class of problems with the property that if one problem can be solved in $o(m)$ time per operation, then all the problems in the class can be solved in $o(m)$ time per operation. Reif

[1] All the logarithms are assumed to be to the base 2 unless explicitly specified otherwise.

mentioned, as examples of complete dynamic problems, the acceptance of a linear-time Turing machine, path systems, the Boolean circuit evaluation problem, unit resolution, and depth first search numbering of a graph. These problems seem to have no better solution in response to dynamic changes than simply running a linear-time off-line algorithm. Westbrook and Tarjan [52] ask if there are sublinear-time algorithms for the fully dynamic maintenance of 2-edge-connected components or whether it is possible to show that this is a complete dynamic problem. We answer this question by showing that this problem can be solved in $O(m^{2/3})$ time per operation. When the graph is planar, the operations can be supported more efficiently in $O(\sqrt{n \log\log n}\,)$ time each. We obtain the same bounds even if we are initially given a nonnull graph, $G_0 = (V_0, E_0)$, and we allow $O(|V_0| + |E_0|)$ preprocessing time. In addition, our data structures can support insertions of new vertices and deletions of disconnected vertices.

We remark that the fully dynamic maintenance of the connected components of a graph differs substantially from the fully dynamic maintenance of the 2-edge-connected components. Indeed, in the first problem a single edge insertion can merge at most two components into one. On the other hand, for the second problem a single edge insertion can produce a cycle in the graph that might combine as many as $\Theta(n)$ 2-edge-connected components into one. As a result, by simply alternating the insertion and deletion of the same edge, we can generate a sequence of operations that at each step changes the number of components by $\Theta(n)$. This also explains why we do not maintain 2-edge-connected components' names: indeed, in such a scenario a *DeleteEdge* operation would have to specify as many as $\Theta(n)$ different new names.

Our techniques combine a variety of graph properties, data structures, and new algorithmic tools. Following the ideas of Frederickson [20], we partition graphs into vertex clusters. However, this by itself is not enough to achieve efficient algorithms for our problems. We find a way to maintain a succinct encoding of each cluster that satisfies nice properties with respect to 2-edge-connected components in the graph. This enables us to answer queries about edge connectivity properties without having to look at the entire graph each time.

The remainder of this paper consists of six sections. In §2 we describe our data structure for connected graphs. Section 3 shows how to detect some particular kind of bridges in our graph. Section 4 deals with the implementation of our operations, while §5 analyzes their time complexity. Section 6 extends the same time and space bounds to unconnected graphs. Finally, in §7 we list some concluding remarks.

2. The data structure for connected graphs. For the sake of simplicity, we first describe algorithms dealing with connected graphs, and we then show how to extend the algorithms to unconnected graphs. We will describe a data structure operating on graphs with bounded degree. It is well known (see, for example, references [20] and [24, p. 132]) that each graph $G = (V, E)$ can be transformed into a graph $G' = (V', E')$, whose vertices have degree no greater than three, and G' preserves the bridges of G. The transformation is as follows. For each vertex u of G of degree $d \geq 4$, where $v_0, v_1, \ldots, v_{d-1}$ are the vertices adjacent to u, replace u with new vertices $u_0, u_1, \ldots, u_{d-1}$, and replace edge (u, v_i) with edge (u_i, v_i), $0 \leq i \leq d-1$. Furthermore, add new edges $(u_i, u_{(i+1) \bmod d})$, $0 \leq i \leq d-1$. Notice that each edge (x, y) in G corresponds to a unique edge (x_i, y_j) in G' (but the converse is not always true since G' may contain the new edges of the form $(u_i, u_{(i+1) \bmod d})$, $0 \leq i \leq d-1$). It is easy to show that given a connected graph G with m edges and n vertices, G' contains at most $O(m)$ edges and vertices, and can be generated in $O(m)$ time. This transformation preserves 2-edge-connected components, as the following lemma shows.

LEMMA 2.1. *Let (x, y) be any edge of G, and let (x_i, y_j) be the corresponding edge in G'. Then (x, y) is a bridge in G if and only if (x_i, y_j) is a bridge in G'.*

Proof. Immediate from the fact that by construction (x, y) lies in a (not necessarily simple) cycle in G if and only if (x_i, y_j) lies in a (not necessarily simple) cycle in G'. □

As a consequence of Lemma 2.1 and the fact that new edges of G' cannot be bridges, we restrict our attention to graphs with $O(m)$ edges and vertices and whose vertices have degree not exceeding three. We obtain efficient algorithms for our problem by partitioning a graph G into clusters and by maintaining a succinct internal representation of each cluster. We do this as follows. We consider any spanning tree T of G and generate a topological partition of T of order k as defined in [20], with k being a positive integer to be chosen later. We recall that a topological partition of T of order k consists of $O(\frac{m}{k})$ vertex clusters of T such that each cluster contains a connected subgraph with between k and $3k-2$ vertices. As shown in [20], this can be accomplished in $O(m)$ time. Throughout the sequence of operations, we maintain for each vertex v of G information about the cluster of the topological partition containing v, in such a way that we can find in constant time the cluster containing a given vertex. This information can be easily initialized and updated during any sequence of our operations, and we will not mention this any further. Furthermore, we maintain the graph $\widetilde{G}$ obtained from G by contracting each cluster $\mathcal{C}$ into a single vertex (also denoted by $\mathcal{C}$). If there are more than two edges between cluster $\mathcal{C}_i$ and $\mathcal{C}_j$ in G, then we represent only two edges between $\mathcal{C}_i$ and $\mathcal{C}_j$ in $\widetilde{G}$. We refer to $\widetilde{G}$ as the *super-graph of G*. Figure 2 shows a graph G together with its super-graph $\widetilde{G}$.

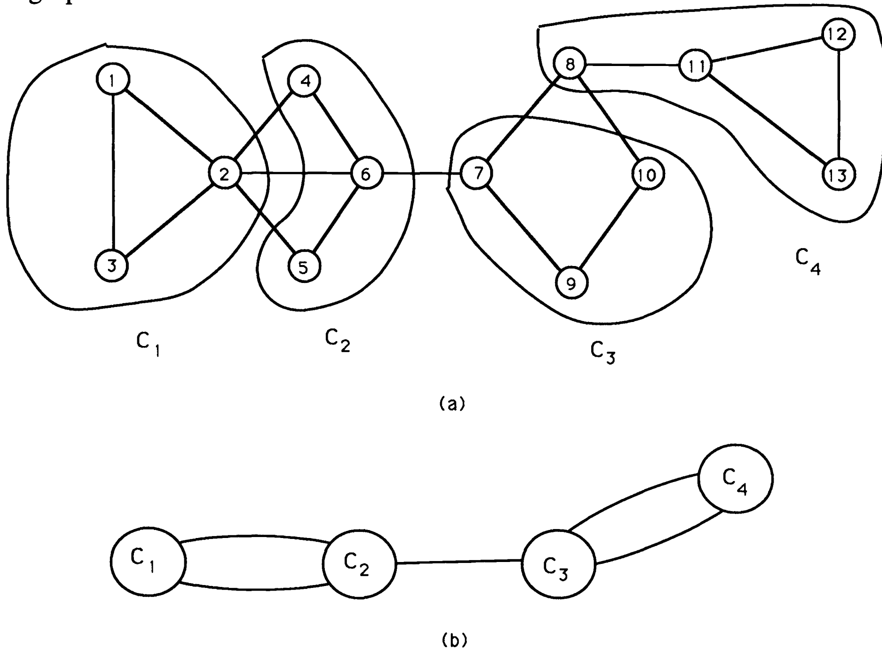


FIG. 2. (a) *A graph $G = (V, E)$ partitioned into clusters*; (b) *the super-graph of G.*

Furthermore, for each cluster $\mathcal{C}$ in the topological partition of T we maintain a *full representation* defined as follows. An edge of G with both endpoints in a cluster $\mathcal{C}$ is said to be *internal* to $\mathcal{C}$, while an edge of G with only one endpoint in $\mathcal{C}$ is said to be *incident* to $\mathcal{C}$. Consider the subgraph induced by cluster $\mathcal{C}$ (i.e., the graph having only vertices and edges internal to $\mathcal{C}$). Call this subgraph $\mathcal{G}(\mathcal{C})$. The *full representation* of cluster $\mathcal{C}$ is a

graph $\mathcal{F}(\mathcal{C})$, which consists of $\mathcal{G}(\mathcal{C})$, plus the clusters adjacent to $\mathcal{C}$ (shrunk into vertices) and the edges incident to $\mathcal{C}$. In other words, the vertex set of $\mathcal{F}(\mathcal{C})$ consists of the vertices internal to $\mathcal{C}$ and the clusters adjacent to $\mathcal{C}$. The edge set of $\mathcal{F}(\mathcal{C})$ is composed of all the edges of G internal and incident to $\mathcal{C}$. Since $\mathcal{C}$ contains at most $O(k)$ vertices, each of which has degree no greater than three, then $\mathcal{F}(\mathcal{C})$ has at most $O(k)$ vertices and edges and can be computed in $O(k)$ time. Figure 3 shows the full representation of a cluster.

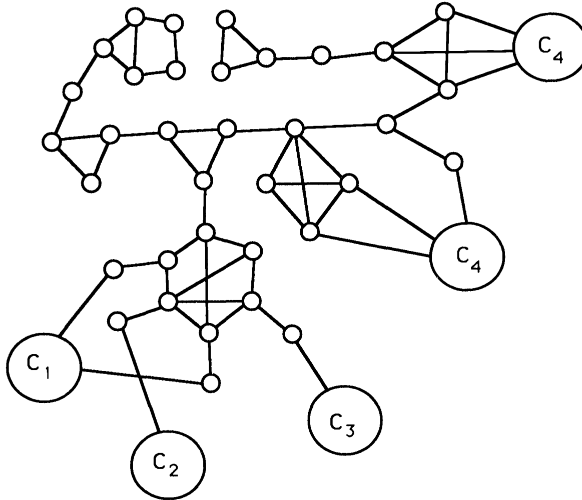


FIG. 3. *The full representation of a cluster C.*

Given the super-graph $\widetilde{G}$ and the full representation of each cluster in the topological partition, we use these data structures to test whether two vertices x and y are in the same 2-edge-connected component, as follows. Denote by $\mathcal{C}_x$ and $\mathcal{C}_y$ the clusters containing, respectively, vertices x and y. Notice that $\mathcal{C}_x$ and $\mathcal{C}_y$ may not be necessarily distinct. Replace in the super-graph $\widetilde{G}$ the vertices $\mathcal{C}_x$ and $\mathcal{C}_y$ with their full representation. That is, replace vertex $\mathcal{C}_x$ and the edges incident to it in $\widetilde{G}$ with $\mathcal{G}(\mathcal{C}_x)$ (the subgraph induced by cluster $\mathcal{C}_x$) and the edges of G incident to cluster $\mathcal{C}_x$. Do exactly the same for cluster $\mathcal{C}_y$. Denote the obtained graph $\mathcal{G}_{x,y}(G)$ and call it *super-graph of G induced by vertices x and y*. Figure 4 shows a graph $G = (V, E)$ and the super-graph induced by two vertices.

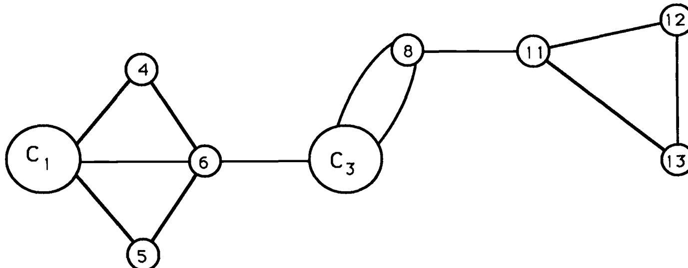


FIG. 4. *The super-graph of the graph in Fig.* 1(a) *induced by vertices* 4 *and* 11.

We now characterize some properties of $\mathcal{G}_{x,y}(G)$.

LEMMA 2.2. *For any two vertices x and y, $\mathcal{G}_{x,y}(G)$ has at most $O(k + (m/k))$ vertices and $O(k+(m^2/k^2))$ edges and can be computed in a total of $O(k)$ time from the super-graph $\widetilde{G}$.*

Proof. Let $\widetilde{G}$ be the super-graph of G. Since there are at most $O(m/k)$ clusters in the topological partition of the original graph G, the number of vertices in $\widetilde{G}$ is $O(m/k)$.

Furthermore, there can be at most two multiple edges between any two vertices of $\widetilde{G}$; consequently, the number of edges in $\widetilde{G}$ is $O(m^2/k^2)$. $\mathcal{G}_{x,y}(G)$ is obtained from $\widetilde{G}$ by replacing at most two of its vertices with the full representation of their corresponding clusters. Since the full representation of a cluster has at most $3k - 2$ vertices, each of degree at most three, this introduces at most $O(k)$ new vertices and edges. As a result, $\mathcal{G}_{x,y}(G)$ has at most $O(k + (m/k))$ vertices and $O(k + (m^2/k^2))$ edges. Furthermore, once the super-graph $\widetilde{G}$ is given, $\mathcal{G}_{x,y}(G)$ can be computed in $O(k)$ time. □

Let x and y be two vertices that are not in the same 2-edge-connected component of G. Then there is a bridge separating x and y in G. Any such bridge e can be of two different types: type (1) if e is either internal to $\mathcal{C}_x$ and $\mathcal{C}_y$ or is an edge between two different clusters and type (2) if e is internal to a cluster $\mathcal{C}$, $\mathcal{C} \neq \mathcal{C}_x$, $\mathcal{C} \neq \mathcal{C}_y$.

LEMMA 2.3. *Let x and y be any two vertices of G. Then the following is true:*

(i) *e is a type (1) bridge in G if and only if the corresponding edge is a bridge in $\mathcal{G}_{x,y}(G)$;*

(ii) *If e is a type (2) bridge separating x and y in G, and e is internal to a cluster $\mathcal{C}$ ($\mathcal{C} \neq \mathcal{C}_x$, $\mathcal{C} \neq \mathcal{C}_y$), then $\mathcal{C}$ is an articulation point in $\mathcal{G}_{x,y}(G)$.*

Proof. To prove (i), we observe that an edge e in $\mathcal{G}_{x,y}(G)$ corresponds either to an internal edge of $\mathcal{C}_x$ or $\mathcal{C}_y$ or to an edge connecting two different clusters. It is easy to see that there is a cycle in $\mathcal{G}_{x,y}(G)$ containing edge e if and only if there is a cycle in G containing the corresponding edge.

We now turn to (ii). If e is a bridge separating x and y in G, then its removal disconnects x and y in G. Since $\mathcal{G}_{x,y}(G)$ can be obtained from G by contracting each cluster except $\mathcal{C}_x$ and $\mathcal{C}_y$ into a single vertex and by reducing to two the number of multiple edges, then the removal of $\mathcal{C}$ must disconnect x and y in $\mathcal{G}_{x,y}(G)$. Therefore, $\mathcal{C}$ must be an articulation point in $\mathcal{G}_{x,y}(G)$. □

By Lemma 2.3, a type (1) bridge can be detected by simply computing the 2-edge-connected components of $\mathcal{G}_{x,y}(G)$. This can be performed by using the off-line linear-time algorithm of Tarjan [47] on $\mathcal{G}_{x,y}(G)$, which requires $O(k + (m^2/k^2))$ time because of Lemma 2.2. The crucial task is, therefore, how to check efficiently whether there is a bridge separating x and y within a cluster $\mathcal{C}$. We will show how to perform this task in the next subsection. First, we need to introduce some new terminology and data structures.

For each cluster $\mathcal{C}$, we maintain its *tree representation* defined as follows. We recall that we denote by $\mathcal{G}(\mathcal{C})$ the subgraph of G induced by cluster $\mathcal{C}$ (i.e., the graph having only vertices and edges internal to $\mathcal{C}$). Compute the 2-edge-connected components of $\mathcal{G}(\mathcal{C})$. They form a tree $\mathcal{T}(\mathcal{C})$, whose edges are the bridges in $\mathcal{G}(\mathcal{C})$. $\mathcal{T}(\mathcal{C})$ is called the *tree representation* of cluster $\mathcal{C}$. As a consequence of any cluster having at most $O(k)$ edges and vertices, the tree representation of a cluster has size at most $O(k)$ and can be computed in $O(k)$ time.

For each cluster $\mathcal{C}$, we root arbitrarily at any vertex the tree representation $\mathcal{T}(\mathcal{C})$, and maintain it as a data structure that supports fast *lowest common ancestor* (in short *lca*) queries [25], [41]. We assume that in a rooted tree edges are directed from the children to the parent. We recall here that given a rooted tree T and two vertices u and v, the lowest common ancestor of u and v, denoted by lca(u, v), is the deepest vertex in T that is ancestor of both u and v. The data structures proposed in [25], [41] answer each lca query in $O(1)$ time after a linear-time preprocessing on the tree. Therefore, in our case,

we need an extra $O(k)$ time to organize $\mathcal{T}(\mathcal{C})$ as a fast lca data structure; but then we can answer any lca query in $\mathcal{T}(\mathcal{C})$ in constant time. Finally, for each vertex v of $\mathcal{T}(\mathcal{C})$, we compute *size*(v) as the size of the subtree rooted at v, and *preorder*(v) as the preorder number of v. For all vertices v in $\mathcal{T}(\mathcal{C})$, *size*(v) and *preorder*(v) can be computed in a total of $O(k)$ time during a preorder traversal of $\mathcal{T}(\mathcal{C})$.

Furthermore, for any two clusters $\mathcal{C}_i$ and $\mathcal{C}_j$, let $E_{i,j} = \{e_1, e_2, \ldots, e_h\}$ be the set of edges of the original graph G between $\mathcal{C}_i$ and $\mathcal{C}_j$. We represent edges in $E_{i,j}$ using two data structures: one for $\mathcal{C}_i$, named $E(\mathcal{C}_i, \mathcal{C}_j)$, and the other for $\mathcal{C}_j$, named $E(\mathcal{C}_j, \mathcal{C}_i)$. $E(\mathcal{C}_i, \mathcal{C}_j)$ stores information about the endpoints of edges in $E_{i,j}$ falling into cluster $\mathcal{C}_j$. $E(\mathcal{C}_j, \mathcal{C}_i)$ is similar. Specifically, $E(\mathcal{C}_i, \mathcal{C}_j)$ contains all the vertices u in $\mathcal{T}(\mathcal{C}_j)$ such that there is an edge in $E_{i,j}$ incident to u. We recall that vertices of $\mathcal{T}(\mathcal{C}_j)$ represent 2-edge-connected components of $\mathcal{G}(\mathcal{C}_j)$, the subgraph of G induced by cluster $\mathcal{C}_j$. Furthermore, we maintain $E(\mathcal{C}_i, \mathcal{C}_j)$ as an ordered set according to the preorder numbering of $\mathcal{T}(\mathcal{C}_j)$. To access and update $E(\mathcal{C}_i, \mathcal{C}_j)$ and $E(\mathcal{C}_j, \mathcal{C}_i)$ efficiently, we maintain them as balanced search trees [2]; the key of vertex v is *preorder*(v).

In summary, throughout our sequence of *Same2EdgeBlock*, *InsertEdge*, and *DeleteEdge* operations we maintain the topological partition into clusters together with the following data structures:

(i) $\widetilde{G}$, the super-graph of G;

(ii) For each cluster $\mathcal{C}$, we maintain its full representation $\mathcal{F}(\mathcal{C})$ and its tree representation $\mathcal{T}(\mathcal{C})$; and

(iii) For each nonempty edge set $E_{i,j}$ between two clusters $\mathcal{C}_i$ and $\mathcal{C}_j$, we maintain the two balanced search trees $E(\mathcal{C}_i, \mathcal{C}_j)$ and $E(\mathcal{C}_j, \mathcal{C}_i)$.

$\mathcal{F}(\mathcal{C})$, $\mathcal{T}(\mathcal{C})$, and all the balanced search trees $E(\mathcal{C}', \mathcal{C})$ for clusters $\mathcal{C}'$ adjacent to $\mathcal{C}$ are referred to as the *data structures pertinent to cluster* $\mathcal{C}$. The following lemma bounds the total time required to initialize all the data structures pertinent to a cluster as well as their space usage.

LEMMA 2.4. *All the data structures pertinent to a cluster $\mathcal{C}$ can be initialized in a total of $O(k)$ time and require $O(k)$ space.*

Proof. The full representation $\mathcal{F}(\mathcal{C})$ of a cluster contains at most $O(k)$ edges and vertices and, therefore, can be computed in $O(k)$ time. On the other hand, the tree representation $\mathcal{T}(\mathcal{C})$ can be obtained by computing the 2-edge-connected components of the subgraph $\mathcal{G}(\mathcal{C})$ induced by cluster $\mathcal{C}$. Since $\mathcal{G}(\mathcal{C})$ has at most $O(k)$ vertices and edges this can be done in $O(k)$ time [47]. Furthermore, the preprocessing involved in computing *size*(v) and *preorder*(v) for each vertex v, as well as in organizing $\mathcal{T}(\mathcal{C})$ as a fast lca data structure requires $O(|\mathcal{T}(\mathcal{C})|) = O(k)$ time [25], [41].

We compute all the balanced search trees $E(\mathcal{C}', \mathcal{C})$, for each cluster $\mathcal{C}'$ neighbor of $\mathcal{C}$, at the same time. We traverse $\mathcal{T}(\mathcal{C})$ in preorder, and we build the balanced search trees by inserting an item in $E(\mathcal{C}', \mathcal{C})$; each time we find an edge between the vertex v of $\mathcal{T}(\mathcal{C})$ we are currently visiting and cluster $\mathcal{C}'$. Since $\mathcal{T}(\mathcal{C})$ is visited in a preorder fashion, items are always inserted at the end of a balanced search tree. Therefore, each balanced search tree can be built in linear time [8]. The total time involved in building all these balanced search trees is

$$O\left(\sum_{\mathcal{C}'} |E(\mathcal{C}', \mathcal{C})|\right) = O(k),$$

since there can be at most a total of $O(k)$ edges of G incident to cluster $\mathcal{C}$.

Consequently, all the data structures pertinent to a cluster $\mathcal{C}$ can be initialized in a total of $O(k)$ time. Obviously, they require $O(k)$ space. □

3. Testing for bridges inside clusters. The data structures given above can be used to check efficiently whether there is a bridge inside a cluster $\mathcal{C}$ that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$.

Consider the super-graph $\widetilde{G}$. Since $\widetilde{G}$ can be obtained from $\mathcal{G}_{x,y}(G)$ by shrinking the two clusters $\mathcal{C}_x$ and $\mathcal{C}_y$ into one super-vertex and by reducing to two the number of multiple edges; $\mathcal{C}$ must be an articulation point separating $\mathcal{C}_x$ and $\mathcal{C}_y$ in $\widetilde{G}$. Thus, removing $\mathcal{C}$ from $\widetilde{G}$ breaks $\widetilde{G}$ into, say, $s(\mathcal{C}) \geq 2$ connected components $W_1, W_2, \ldots, W_{s(\mathcal{C})}$. Without loss of generality, let W_1 be the connected component containing $\mathcal{C}_x$, and let W_2 be the connected component containing $\mathcal{C}_y$. Denote by $\mathcal{Z}_{\mathcal{C}}^{h,1}, \mathcal{Z}_{\mathcal{C}}^{h,2}, \ldots, \mathcal{Z}_{\mathcal{C}}^{h,r_h(\mathcal{C})}$ the clusters previously adjacent to $\mathcal{C}$ and now in connected component W_h, $1 \leq h \leq s(\mathcal{C})$. Let $\mathcal{X}_{\mathcal{C}}^i = \mathcal{Z}_{\mathcal{C}}^{1,i}$, $i = 1, 2, \ldots r_1(\mathcal{C}) \equiv p(\mathcal{C})$ (i.e., $\mathcal{X}_{\mathcal{C}}^1, \mathcal{X}_{\mathcal{C}}^2, \ldots, \mathcal{X}_{\mathcal{C}}^{p(\mathcal{C})}$ are the clusters previously adjacent to $\mathcal{C}$ and now in the same connected component as $\mathcal{C}_x$). Similarly, let $\mathcal{Y}_{\mathcal{C}}^j = \mathcal{Z}_{\mathcal{C}}^{2,j}$, $j = 1, 2, \ldots r_2(\mathcal{C}) \equiv q(\mathcal{C})$ (i.e., $\mathcal{Y}_{\mathcal{C}}^1, \mathcal{Y}_{\mathcal{C}}^2, \ldots, \mathcal{Y}_{\mathcal{C}}^{q(\mathcal{C})}$ are the clusters previously adjacent to $\mathcal{C}$ and now in the same connected component as $\mathcal{C}_y$). Notice that $p(\mathcal{C}) \geq 1$, $q(\mathcal{C}) \geq 1$, $r_h(\mathcal{C}) \geq 1$, $3 \leq h \leq s(\mathcal{C})$, and $p(\mathcal{C}) + q(\mathcal{C}) + \sum_{h=3}^{s(\mathcal{C})} r_h(\mathcal{C}) = d(\mathcal{C})$, where $d(\mathcal{C})$ is the number of clusters adjacent to $\mathcal{C}$, also called the *external degree* of cluster $\mathcal{C}$.

LEMMA 3.1. *Given $\widetilde{G}$, all the clusters $\mathcal{C}$ that are articulation points separating x and y in $\mathcal{G}_{x,y}(G)$, as well as their neighboring clusters $\mathcal{X}_{\mathcal{C}}^i$, $1 \leq i \leq p(\mathcal{C})$, $\mathcal{Y}_{\mathcal{C}}^j$, $1 \leq j \leq q(\mathcal{C})$, and $\mathcal{Z}_{\mathcal{C}}^{h,t}$, $3 \leq h \leq s(\mathcal{C})$, $1 \leq t \leq r_h(\mathcal{C})$, can be found in a total of $O(m^2/k^2)$ time.*

Proof. Compute the biconnected components of $\widetilde{G}$ using the algorithm of Tarjan [47], and subsequently construct the block tree of $\widetilde{G}$ (as defined for instance in [24]). We recall here that a block tree of a graph is composed of square nodes (corresponding to vertices in the graph) and round nodes (corresponding to biconnected components in the graph): whenever a vertex belongs to a given biconnected component, there is a tree edge between the corresponding square and round nodes in the block tree. As a consequence, each path in a block tree alternates between square and round nodes, and the block tree has size bounded by the total number of square nodes, i.e., by the number of vertices in the original graph. The construction of the block tree is immediate once the biconnected components have been found. Since $\widetilde{G}$ has $O(m/k)$ vertices and $O(m^2/k^2)$ edges, its biconnected components and its block tree can be computed in $O(m^2/k^2)$ time. Furthermore, the block tree has size $O(m/k)$. Due to the property of block trees, all the clusters $\mathcal{C}$ that are articulation points separating x and y in $\mathcal{G}_{x,y}(G)$ can be found as the square nodes in the path between $\mathcal{C}_x$ and $\mathcal{C}_y$ in the block tree of $\widetilde{G}$. Furthermore, each cluster $\mathcal{C}'$ can be adjacent to at most two clusters that are articulation points separating x and y in $\mathcal{G}_{x,y}(G)$. Consequently, each cluster $\mathcal{C}'$ can be in each of the sets $\{\mathcal{X}_{\mathcal{C}}^i\}_{1 \leq i \leq p(\mathcal{C})}$, $\{\mathcal{Y}_{\mathcal{C}}^j\}_{1 \leq j \leq q(\mathcal{C})}$, and $\{\mathcal{Z}_{\mathcal{C}}^{h,t}\}_{3 \leq h \leq s(\mathcal{C}), 1 \leq t \leq r_h(\mathcal{C})}$ at most once for all possible choices of the articulation point $\mathcal{C}$ in $\widetilde{G}$. As a result, all the neighbors $\mathcal{X}_{\mathcal{C}}^i$, $\mathcal{Y}_{\mathcal{C}}^j$, and $\mathcal{Z}_{\mathcal{C}}^{h,t}$ of all these clusters $\mathcal{C}$ can be found in a total of $O(m^2/k^2)$ time by visiting both $\widetilde{G}$ and its tree. □

We now show how to check whether there is a bridge internal to a cluster $\mathcal{C}$ that is an articulation point in $\mathcal{G}_{x,y}(G)$. Assume that a vertex v in $\mathcal{T}(\mathcal{C})$, the tree representation of $\mathcal{C}$, is colored *red* if there is an edge in G between a vertex in cluster $\mathcal{X}_{\mathcal{C}}^i$, $1 \leq i \leq p(\mathcal{C})$, and v. Similarly, assume that a vertex v in $\mathcal{T}(\mathcal{C})$ is colored *black* if there is an edge in G between a vertex in cluster $\mathcal{Y}_{\mathcal{C}}^j$, $1 \leq j \leq q(\mathcal{C})$, and v. The red vertices are all the vertices in $E(\mathcal{X}_{\mathcal{C}}^i, \mathcal{C})$, $1 \leq i \leq p(\mathcal{C})$, and the black vertices are all the vertices in $E(\mathcal{Y}_{\mathcal{C}}^j, \mathcal{C})$, $1 \leq j \leq q(\mathcal{C})$. Similarly, we denote the clusters $\mathcal{X}_{\mathcal{C}}^i$, $1 \leq i \leq p(\mathcal{C})$ as *red clusters*, and the clusters $\mathcal{Y}_{\mathcal{C}}^j$, $1 \leq j \leq q(\mathcal{C})$ as *black clusters*. The other clusters, $\mathcal{Z}_{\mathcal{C}}^{h,t}$, $3 \leq h \leq s(\mathcal{C})$,

$1 \leq t \leq r_h(\mathcal{C})$, are referred to as *W-clusters*. Notice that since $p(\mathcal{C}) \geq 1$ and $q(\mathcal{C}) \geq 1$, we have at least one black vertex and one red vertex in $\mathcal{T}(\mathcal{C})$.

We need the following new terminology. Given the tree representation $\mathcal{T}(\mathcal{C})$ of cluster $\mathcal{C}$, and an edge e in $\mathcal{T}(\mathcal{C})$, let T_1^e and T_2^e be the two trees obtained by removing e from $\mathcal{T}(\mathcal{C})$. We recall that $W_1, W_2, \ldots, W_{s(\mathcal{C})}$ are the connected components left in the super-graph $\widetilde{G}$ after the removal of cluster $\mathcal{C}$. We say that a connected component W_h, $1 \leq h \leq s(\mathcal{C})$, is *compatible with* edge e if all the edges between clusters $\mathcal{Z}_{\mathcal{C}}^{h,t}$, $1 \leq t \leq r_h(C)$, are incident to either T_1^e or T_2^e, but not to both.

LEMMA 3.2. *Let $\mathcal{C}$ be a cluster that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$. There is a bridge inside cluster $\mathcal{C}$ that separates vertices x and y in G if and only if there is an edge e in $\mathcal{T}(\mathcal{C})$ that satisfies the following two conditions:*

(i) *The removal of e separates red and black vertices; and*

(ii) *Each connected component W_h, $3 \leq h \leq s(\mathcal{C})$, is compatible with e.*

Proof. By definition of the clusters $\mathcal{X}_{\mathcal{C}}^i$ and $\mathcal{Y}_{\mathcal{C}}^j$, every path between x and y in G must have the following structure. It starts with a path outside $\mathcal{C}$ ending at a vertex of $\mathcal{C}$ that corresponds to a red vertex in $\mathcal{T}(\mathcal{C})$. It ends with a path outside $\mathcal{C}$ starting at a vertex of $\mathcal{C}$ that corresponds to a black vertex in $\mathcal{T}(\mathcal{C})$.

Thus, there is an edge e in $\mathcal{T}(\mathcal{C})$ for which conditions (i) and (ii) hold if and only if the removal of the edge corresponding to e in G separates x and y. Condition (i) holds if and only if any path in $\mathcal{C}$ between a vertex corresponding to a red vertex and a vertex corresponding to a black vertex contains e. Condition (ii) holds if and only if there is no detour avoiding e through component W_h, $3 \leq h \leq s(\mathcal{C})$. □

Figure 5 shows an example of an edge that satisfies both conditions of Lemma 3.2.

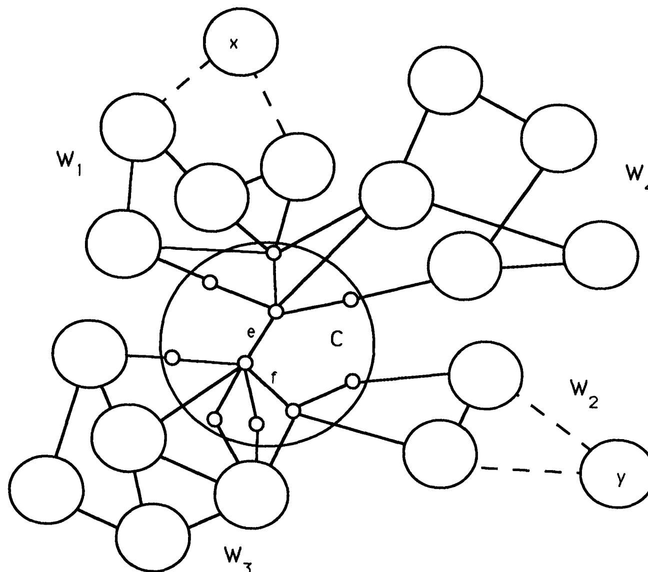


FIG. 5. *A cluster C, its tree representation, and its neighboring clusters. Edge f satisfies only condition* (i) *of Lemma* 4.2, *while edge e satisfies both conditions* (i) *and* (ii).

We remark that, in order to check conditions (i) and (ii) of Lemma 3.2, we cannot even afford to enumerate all the red and black vertices explicitly. Indeed this can take as much as $\Omega(k)$ time for each cluster $\mathcal{C}$ we need to consider. Since we may have to perform this task in $\Omega(m/k)$ different clusters, these times would sum up to a total of $\Omega(m)$. We check the two conditions of Lemma 3.2 independently, and then intersect the solutions of these two computations.

We first describe how to check efficiently condition (i). First we need few technical lemmas. Let $\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell$, $\ell > 0$, be any ℓ clusters adjacent to cluster $\mathcal{C}$. Then define by $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell)$ the deepest vertex in $\mathcal{T}(\mathcal{C})$ such that all the edges between $\mathcal{A}_i$, $1 \leq i \leq \ell$, and $\mathcal{C}$ are incident below $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell)$.

LEMMA 3.3. *The vertex $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell)$ can be found in $O(\ell)$ time.*

Proof. For $1 \leq i \leq \ell$, find λ_i, the lca of all the vertices in $E(\mathcal{A}_i, \mathcal{C})$. Since $E(\mathcal{A}_i, \mathcal{C})$ is sorted according to the preorder numbering of $\mathcal{T}(\mathcal{C})$, this can be done by first finding the leftmost and rightmost vertices u_i and v_i in $E(\mathcal{A}_i, \mathcal{C})$, and then by computing $\lambda_i = lca(u_i, v_i)$. Then $\lambda(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_\ell)$ can be found as the lca of all the λ_i's by performing at most $O(\ell)$ lca queries. The total time required can be bounded as follows. Finding the leftmost and rightmost item in a balanced search tree $E(\mathcal{A}_i, \mathcal{C})$ requires $O(\log |E(\mathcal{A}_i, \mathcal{C})|)$ time. However, this can be performed in constant time if we maintain extra pointers from the root of each balanced search tree to the leftmost and rightmost items in the tree. The time required to perform an lca query in $\mathcal{T}(\mathcal{C})$ is $O(1)$. Since we have at most a total of $O(\ell)$ queries, the time spent in finding the leftmost and rightmost vertices in all the balanced search trees $E(\mathcal{A}_i, \mathcal{C})$, $1 \leq i \leq \ell$, and in performing all the lca queries will be $O(\ell)$. Consequently, the total time required is $O(\ell)$. □

Define the *top-red* vertex ρ as $\lambda(\mathcal{X}_{\mathcal{C}}^1, \mathcal{X}_{\mathcal{C}}^2, \ldots, \mathcal{X}_{\mathcal{C}}^{p(\mathcal{C})})$. Notice that ρ is the lca of all the red vertices in $\mathcal{T}(\mathcal{C})$. Similarly, define the *top-black* vertex β as $\lambda(\mathcal{Y}_{\mathcal{C}}^1, \mathcal{Y}_{\mathcal{C}}^2, \ldots, \mathcal{Y}_{\mathcal{C}}^{q(\mathcal{C})})$, that is, as the lca of all the black vertices in $\mathcal{T}(\mathcal{C})$. The following corollary is a consequence of Lemma 3.3.

COROLLARY 3.1. *The top-red and the top-black vertices ρ and β in $\mathcal{T}(\mathcal{C})$ can be found in $O(p(\mathcal{C}) + q(\mathcal{C}))$ time.*

After locating the top-red and top-black vertices ρ and β, we compute in $O(1)$ time $v = lca(\rho, \beta)$. We now have four different cases depending on the relative positions of the three vertices v, ρ, and β in $\mathcal{T}(\mathcal{C})$.

LEMMA 3.4. *Let ρ be the top-red vertex, β the top-black vertex, and let $v = lca(\rho, \beta)$. Let $\pi_{\rho,\beta}$ be the path in $\mathcal{T}(\mathcal{C})$ between ρ and β. Then*

(1) If $v = \rho = \beta$, no edge in $\mathcal{T}(\mathcal{C})$ can separate black and red vertices;

(2) If $v \neq \rho$ and $v \neq \beta$, then all the edges in $\pi_{\rho,\beta}$ (and only them) separate black and red vertices;

(3) If $v = \beta \neq \rho$, an edge e separates black and red vertices if and only if e is in $\pi_{\rho,\beta}$ and there are no black vertices below e;

(4) If $v = \rho \neq \beta$, an edge e separates black and red vertices if and only if e is in $\pi_{\rho,\beta}$ and there are no red vertices below e.

Proof. We exploit the following fact. The deletion of an edge splits the tree $\mathcal{T}(\mathcal{C})$ into two trees. Since there is at least one red vertex and one black vertex in $\mathcal{T}(\mathcal{C})$, an edge separates black and red vertices if and only if its removal leaves all the black vertices in one tree and all the red vertices in the other tree. As a result, an edge that separates either two black vertices or two red vertices cannot separate black and red vertices. We now analyze the four different cases.

In case (1) $v = \rho = \beta$. By definition of lca we must have two (not necessarily distinct) red vertices, r_1 and r_2, and two (not necessarily distinct) black vertices, b_1 and b_2, such

that the path π_R between r_1 and r_2 and the path π_B between b_1 and b_2 intersect in v. Notice, however, that if $b_1 = b_2$ [$r_1 = r_2$], then it must be $v = b_1 = b_2$ [$v = r_1 = r_2$]. Assume by contradiction that there is an edge e in $\mathcal{T}(\mathcal{C})$ whose removal separates red and black vertices. Clearly e cannot be in π_B; otherwise two black vertices b_1 and b_2 will be separated. For the same reason e cannot be in π_R. But this implies that after the removal of e, we have that a red vertex (r_1 or r_2) is still reachable from a black vertex (b_1 or b_2), a contradiction.

We now show that in cases (2), (3), and (4) only edges in $\pi_{\rho,\beta}$ can possibly separate red and black vertices. Indeed the removal of an edge not in $\pi_{\rho,\beta}$ leaves β and ρ in the same tree. The same argument given in case (1) shows that such an edge cannot separate red and black vertices.

Now consider case (2). All the red vertices are in the subtree rooted at ρ, and all the black vertices are in the subtree rooted at β. Since $v = lca(\rho, \beta)$ is different from both ρ and β, then all the edges in $\pi_{\rho,\beta}$ (and as shown before only them) separate red and black vertices.

As for case (3), let $e = (a, b)$ be any edge in $\pi_{\rho,\beta}$, with b being the parent of a. If there is no black vertex in the subtree rooted at a, then the removal of e will separate red and black vertices. Assume now there is a black vertex, say b_1, in the subtree rooted at a. Since β is the lca of all the black vertices and it is a proper ancestor of a, there must be another black vertex $b_2 \neq b_1$ outside the subtree rooted at a. Therefore, the removal of e will split $\mathcal{T}(\mathcal{C})$ into two pieces, each of which contains at least one black vertex. As a result, e cannot separate red and black vertices.

Case (4) is analogous to case (3). □

Conditions (1) and (2) of Lemma 3.4 are both easy to check once ρ, β, and v are available; furthermore, they immediately give all edges separating red and black vertices. We still need to explain how to check conditions (3) and (4). Without loss of generality, we restrict our attention to condition (3). We do it separately for each $\mathcal{Y}_\mathcal{C}^j$ cluster inducing black vertices in $\mathcal{T}(\mathcal{C})$. The idea underlying a fast algorithm to accomplish this task is to use a kind of binary search instead of checking all the possible black vertices.

LEMMA 3.5. *Let $\mathcal{C}'$ be any cluster adjacent to $\mathcal{C}$, and let v be any vertex of $\mathcal{T}(\mathcal{C})$. Then we can check whether there are edges between $\mathcal{C}'$ and $\mathcal{C}$ incident below v in $O(\log k)$ time.*

Proof. By definition of $E(\mathcal{C}', \mathcal{C})$, there are edges between $\mathcal{C}'$ and $\mathcal{C}$ below v if and only if there are vertices of $E(\mathcal{C}', \mathcal{C})$ below v. Because of preorder numbering, a vertex u in $E(\mathcal{C}', \mathcal{C})$ is in the subtree of $\mathcal{T}(\mathcal{C})$ rooted at v if and only if the preorder number of u satisfies $preorder(v) \leq preorder(u) \leq preorder(v) + size(v) - 1$. Consequently, we have to check whether the balanced search tree $E(\mathcal{C}', \mathcal{C})$ contains at least one item in the range $[preorder(v), preorder(v)+size(v)-1]$. This range query can be accomplished by finding the smallest item of the balanced search tree $E(\mathcal{C}', \mathcal{C})$ greater than or equal to $preorder(v)$, which can be done in $O(\log |E(\mathcal{C}', \mathcal{C})|) = O(\log k)$ time. Then $E(\mathcal{C}', \mathcal{C})$ contains at least one item in the range $[preorder(v), preorder(v)+size(v)-1]$ if and only if the item found is less than or equal to $preorder(v)+size(v)-1$. □

LEMMA 3.6. *Given a tree representation $\mathcal{T}(\mathcal{C})$ of a cluster $\mathcal{C}$ that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$, we can find whether red and black vertices can be separated in $\mathcal{T}(\mathcal{C})$ in $O((p(\mathcal{C}) + q(\mathcal{C})) \log k)$ time.*

Proof. The three vertices ρ, β, and v can be computed in $O(p(\mathcal{C}) + q(\mathcal{C}))$ time by Corollary 3.1. By Lemma 3.4, we have to check only four possible conditions. Conditions (1) and (2) can be checked in constant time and require no further work.

We show how to perform efficiently the test implied in condition (3) for each $E(\mathcal{Y}_\mathcal{C}^j, \mathcal{C})$, $1 \leq j \leq q(\mathcal{C})$. Notice that in order to check whether red and black vertices

can be separated in $\mathcal{T}(\mathcal{C})$, it is enough to check that there are no black vertices in the subtree of $\mathcal{T}(\mathcal{C})$ rooted at ρ. By Lemma 3.5, we can check in $O(\log k)$ time whether there are edges between $\mathcal{Y}_{\mathcal{C}}^j$ and $\mathcal{C}$. Since we have to test this for all the possible clusters $\mathcal{Y}_{\mathcal{C}}^j$, $1 \leq j \leq q(\mathcal{C})$, condition (3) requires a total of $O(q(\mathcal{C}) \log k)$ time.

Condition (4) of Lemma 3.4 can be checked with the same algorithm, but this time using the clusters $\mathcal{X}_{\mathcal{C}}^i$ instead, $1 \leq i \leq p(\mathcal{C})$. This requires a total of $O(p(\mathcal{C}) \log k)$ time.

In summary, we can check whether there is an edge separating red and black vertices in $\mathcal{T}(\mathcal{C})$ in a total of $O((p(\mathcal{C}) + q(\mathcal{C})) \log k)$ time. □

By Lemma 3.6, we can check in $O((p(\mathcal{C}) + q(\mathcal{C})) \log k)$ time whether black and red vertices can be separated in cluster $\mathcal{C}$, i.e., whether there is an edge of $\mathcal{T}(\mathcal{C})$ that satisfies condition (i) of Lemma 3.2. If red and black vertices cannot be separated, then by Lemma 3.2 there cannot be any bridge inside cluster $\mathcal{C}$ separating vertices x and y in G.

Otherwise, if red and black vertices can be separated, we check condition (ii) as follows. Let ρ be the top-red vertex and let β be the top-black vertex in $\mathcal{T}(\mathcal{C})$. If red and black vertices can be separated, it means that we are either in case (2), (3), or (4) of Lemma 3.4. As shown in Lemma 3.4, $\pi_{\rho,\beta}$, the path between vertices ρ and β in $\mathcal{T}(\mathcal{C})$, contains all the possible edges that separate red and black vertices and for which condition (ii) of Lemma 3.2 has to be checked.

We first consider case (3) of Lemma 3.4. We consider each connected component W_h, $3 \leq h \leq s(\mathcal{C})$. We compute the lca of all the vertices in $E(\mathcal{Z}_{\mathcal{C}}^{h,t}, \mathcal{C})$, $1 \leq t \leq r_h(\mathcal{C})$, and call it γ_h. By Lemma 3.3, this can be done in $O(r_h(\mathcal{C}))$ time. Thus, in a total of

$$O\left(\sum_{h=3}^{s(\mathcal{C})} r_h(\mathcal{C})\right) = O(d(\mathcal{C}))$$

time, we can compute all the vertices $\gamma_3, \gamma_4, \ldots, \gamma_{s(\mathcal{C})}$. If some γ_h, $3 \leq h \leq s(\mathcal{C})$, is not in the path between ρ and the root of $\mathcal{T}(\mathcal{C})$, then the removal of any edge in $\pi_{\rho,\beta}$ leaves all the vertices in $E(\mathcal{Z}_{\mathcal{C}}^{h,t}, \mathcal{C})$, $3 \leq t \leq r_h(\mathcal{C})$, on the same side. Therefore, the corresponding component W_h is compatible with any edge in the path $\pi_{\rho,\beta}$. We examine all the γ_h vertices and discard the vertices not in the path between ρ and the root. Since we can test in $O(1)$ time whether a vertex γ_h is in the path between ρ and the root by simply checking whether $\gamma_h = lca(\rho, \gamma_h)$, this step can be accomplished in a total of $O(s(\mathcal{C})) = O(d(\mathcal{C}))$ time. At the end of this step, we are left with $r \leq s(\mathcal{C})$ vertices $\gamma_{h_1}, \gamma_{h_2}, \ldots, \gamma_{h_r}$ in the path between ρ and the root of $\mathcal{T}(\mathcal{C})$, corresponding to the connected components for which we have still to check condition (ii) of Lemma 3.2. We sort the vertices γ_{h_i}, $1 \leq i \leq r$, in $O(r \log r)$ time according to their preorder numbering in $\mathcal{T}(\mathcal{C})$. Notice that $r \leq s(\mathcal{C}) \leq d(\mathcal{C})$. Furthermore, r is at most the total number of edges incident to cluster $\mathcal{C}$: $r \leq k$. Hence, $r \leq \min\{d(\mathcal{C}), k\}$ and $O(r \log r) = O(d(\mathcal{C}) \log k)$. Let $\gamma'_1, \gamma'_2, \ldots, \gamma'_r$ be the resulting sorted list, with $preorder(\gamma'_i) \geq preorder\ (\gamma'_{i+1})$. Let $\gamma'_0 = \rho$. For $0 \leq i \leq r$, let e'_i be the edge of $\mathcal{T}(\mathcal{C})$ from γ'_i to its parent, if γ'_i is not the root, and let it be undefined otherwise. Let W'_i be the connected component corresponding to γ'_i, and let $\mathcal{Z}_{\mathcal{C}}^{i',t}$, $1 \leq t \leq r_{i'}(\mathcal{C})$, be the clusters in W'_i adjacent to $\mathcal{C}$. The following lemma states that in our search for edges satisfying condition (ii) of Lemma 3.2, without any loss of generality, we can restrict our attention to the edges $e'_1, e'_2, \ldots, e'_r$.

LEMMA 3.7. *Let β be a proper ancestor of ρ. There is an edge of $\mathcal{T}(\mathcal{C})$ satisfying both conditions* (i) *and* (ii) *of Lemma* 3.2 *if and only if at least one edge e'_i, $0 \leq i \leq r$, satisfies conditions* (i) *and* (ii).

Proof. Assume there is an edge of $\mathcal{T}(\mathcal{C})$, say, $e = (a, b)$, with b, being the parent of a, satisfying both conditions (i) and (ii) of Lemma 3.4. By Lemma 3.4, the two vertices

a and b must be in the path $\pi_{\rho,\beta}$. Let $T_1^{(a,b)}$ be the subtree of $\mathcal{T}(\mathcal{C})$ rooted at a, and $T_2^{(a,b)} = \mathcal{T}(\mathcal{C}) - \{T_1^{(a,b)} \cup (a,b)\}$. Let $\gamma'_0, \gamma'_1, \ldots, \gamma'_r$ be the sorted list of vertices in the path between ρ and the root as defined above. Let $\gamma'_0, \gamma'_1, \ldots, \gamma'_\ell$, $0 \leq \ell \leq r$, be all such vertices in the path between ρ and a (ρ and a included). Since e satisfies condition (ii), then each W'_i, $1 \leq i \leq r$, is compatible with e. This means that all the edges between $W'_1, W'_2, \ldots, W'_\ell$, and $\mathcal{C}$ are incident to $T_1^{(a,b)}$, and that all the edges between $W'_{\ell+1}, W'_{\ell+2}, \ldots, W'_r$ and $\mathcal{C}$ are incident to $T_2^{(a,b)}$. By definition of the γ'_i's, the edges between $W'_1, W'_2, \ldots, W'_\ell$, and $\mathcal{C}$ are all incident to vertices of $\mathcal{T}(\mathcal{C})$ below γ'_ℓ. Therefore, each W'_i, $1 \leq i \leq r$, is compatible with e'_ℓ, too. □

We examine the vertices γ'_i, $1 \leq i \leq r$, according to decreasing preorder numbering, i.e., starting with γ'_1. We maintain a vertex called *CANDIDATE*, satisfying the following invariant:

(I1) Before examining vertex γ'_i, $1 \leq i \leq r$, *CANDIDATE*$= \gamma'_j$, $0 \leq j \leq i-1$, where either γ'_j is the root of $\mathcal{T}(\mathcal{C})$ or γ'_j is the deepest vertex such that $W'_1, W'_2, \ldots, W'_{i-1}$ are all compatible with e'_j (the edge from γ'_j to its parent).

At the beginning, $i = 1$, *CANDIDATE*$= \gamma'_0 = \rho$ and invariant (I1) trivially holds. We describe inductively how to examine vertex γ'_i, $1 \leq i \leq r$. Let *CANDIDATE*$= \gamma'_j$, $0 \leq j \leq i-1$, while examining vertex γ'_i. If $\gamma'_i = \gamma'_j$, then we set i to $i+1$. Since W'_i is compatible with $e'_i = e'_j$, invariant (I1) still holds. Otherwise, we check whether all the balanced search trees $E(\mathcal{Z}_C^{i',t}, \mathcal{C})$, $1 \leq t \leq r_{i'}(\mathcal{C})$, have vertices in the subtree rooted at γ'_j. By Lemma 3.5, this can be accomplished in a total of $O(r_{i'}(\mathcal{C}) \log k)$ time by means of $r_{i'}(\mathcal{C})$ range queries. If $E(\mathcal{Z}_C^{i',t}, \mathcal{C})$, $1 \leq t \leq r_{i'}(\mathcal{C})$, have no vertices in the subtree rooted at γ'_j, then the edge e'_j from γ'_j to its parent in $\mathcal{T}(\mathcal{C})$ is compatible with W'_i. Therefore, γ'_j is still the deepest vertex such that $W'_1, W'_2, \ldots, W'_i$ are all compatible with e'_j. Again we set i to $i+1$, and invariant (I1) still holds. Otherwise, there is at least one vertex in the balanced search trees $E(\mathcal{Z}_C^{i',t}, \mathcal{C})$, $1 \leq t \leq r_{i'}(\mathcal{C})$, that is, in the subtree rooted at γ_j. This implies that all the edges $e'_j, e'_{j+1}, \ldots, e'_{i-1}$ (corresponding to γ'_j and all the vertices examined after γ'_j) are not compatible with W'_i. However, either γ'_i is the root or edge e'_i is now compatible with $W'_1, W'_2, \ldots, W'_i$. Consequently, we set *CANDIDATE*$= \gamma'_i$ and $i = i+1$.

This shows that invariant (I1) is maintained after each vertex γ'_i is examined. Let *CANDIDATE* $= \gamma'$ after all the vertices have been examined. Let e' be the edge from γ' to its parent. Because of invariant (I1), e' is the deepest edge in $\{e'_0, e'_1, \ldots, e'_r\}$ that satisfies condition (ii) of Lemma 3.2. Notice that if γ' is the root of $\mathcal{T}(\mathcal{C})$, then e' is not defined; but in this case by Lemma 3.7 no edge of $\mathcal{T}(\mathcal{C})$ satisfies condition (ii). If γ' is not the root, we check whether there is an edge that satisfies both conditions (i) and (ii) as follows.

If γ' is an ancestor of β, then e' is not in $\pi_{\rho,\beta}$ and, therefore, cannot satisfy condition (i). This also implies that the edges in $\{e'_0, e'_1, \ldots, e'_r\}$ above e' cannot be in $\pi_{\rho,\beta}$ and, consequently, do not satisfy condition (i). Since by invariant (I1) e' is the deepest edge in $\{e'_0, e'_1, \ldots, e'_r\}$ satisfying condition (ii), the edges in $\{e'_0, e'_1, \ldots, e'_r\}$ below e' do not satisfy condition (ii). Therefore, no edge in $\{e'_0, e'_1, \ldots, e'_r\}$ satisfies both conditions (i) and (ii). By Lemma 3.2, no edge in $\mathcal{T}(\mathcal{C})$ satisfies both conditions (i) and (ii), and by Lemma 3.2 there is no bridge in $\mathcal{C}$ separating x and y in G.

Otherwise, γ' is a proper descendant of β. By invariant (I1) e' is the deepest edge in $\{e'_0, e'_1, \ldots, e'_r\}$ satisfying condition (ii). To check whether e' also satisfies condition (i), we have to check that there are no black vertices in the subtree rooted at γ'. By Lemma 3.5, this can be accomplished in $O(q(\mathcal{C}) \log k)$ time. If there are no black vertices below

γ', then e' satisfies both conditions (i) and (ii), and by Lemma 3.2 the corresponding edge of $\mathcal{C}$ is a bridge separating x and y in G. Otherwise, there is at least one black vertex in the subtree rooted at γ', and, therefore, e' cannot satisfy condition (i). Consequently, edges in $\{e'_0, e'_1, \ldots, e'_r\}$ above e' do not satisfy condition (i) either. By Lemma 3.7, no edge in $\mathcal{T}(\mathcal{C})$ satisfies both conditions (i) and (ii), and by Lemma 3.2 there is no bridge in $\mathcal{C}$ separating x and y in G.

This shows how to deal with case (3) of Lemma 3.4. Case (4) is analogous. Case (2) can be dealt with in a very similar fashion. The only difference is that now while searching for an edge satisfying condition (ii), we have to consider separately the two paths π_ρ between ρ and the root and π_β between β and the root. By applying the above algorithm at most twice, each time on a different path, we are able to check whether there is an edge in $\pi_{\rho,\beta}$ satisfying both conditions (i) and (ii) of Lemma 3.2.

LEMMA 3.8. *Given a cluster $\mathcal{C}$ that is an articulation point separating x and y in $\mathcal{G}_{x,y}(G)$, we can find whether $\mathcal{C}$ contains a bridge separating x and y in G in $O(d(\mathcal{C}) \log k)$ time, where $d(\mathcal{C})$ is the external degree of cluster $\mathcal{C}$.*

Proof. We analyze the time complexity of the above algorithm as follows. Condition (i) of Lemma 3.2 can be checked in $O((p(C) + q(\mathcal{C})) \log k)$ by Lemma 3.6.

We now bound the time required to check condition (ii). As explained before, the preprocessing required to find and sort the vertices $\gamma'_1, \gamma'_2, \ldots, \gamma'_r$ is $O(d(C) \log k)$. The time involved in examining vertex γ'_i for some i, $1 \le i \le r$, is the time spent in the range queries in the balanced search trees, which is $O(r_{i'}(\mathcal{C}) \log k)$ by Lemma 3.5. Consequently, the total time required to examine the $r \le s(C)$ vertices γ'_i throughout the algorithm is

$$O\left(\sum_{i=1}^{s(\mathcal{C})} r_i(\mathcal{C}) \log k\right) = O(d(\mathcal{C}) \log k).$$

We now have to bound the time required to check whether there is an edge satisfying both conditions (i) and (ii). This involves performing range queries to test whether there is any black or red vertex below the vertex γ' defined above. By Lemma 3.5, this can be implemented in $O((p(C) + q(C)) \log k) = O(d(\mathcal{C}) \log k)$ time.

Since case (2) is the most expensive part of Lemma 3.4, and it requires applying at most twice the above algorithm, we can check whether there is a bridge internal to a cluster $\mathcal{C}$ in a total of $O(d(\mathcal{C}) \log k)$ time. □

4. Implementing the operations. We perform a *Same2EdgeBlock*(x,y) operation as follows. We first compute $\mathcal{G}_{x,y}(G)$, the super-graph induced by vertices x and y. By Lemma 2.2, it can be computed in $O(k)$ time from $\widetilde{G}$. We then find the biconnected components of $\mathcal{G}_{x,y}(G)$ by using the linear-time algorithm of Tarjan [47]. Because of Lemma 2.2 this requires $O(k + (m^2/k^2))$ time in the worst case. If we find a bridge separating x and y in $\mathcal{G}_{x,y}(G)$, then we stop and declare x and y not to be in the same 2-edge-connected component of G by returning *false*. This is correct because of Lemma 2.3. Otherwise, we have still to check whether there is a bridge separating x and y in G that is internal to some cluster $\mathcal{C}$. By Lemma 2.3, we can restrict our attention to clusters that are articulation points separating x and y in $\mathcal{G}_{x,y}(G)$. We proceed as follows.

Let $\mathcal{S}$ be the set of clusters such that (i) they are articulation points in $\mathcal{G}_{x,y}(G)$, and (ii) they separate x and y. Notice that in the worst case, $\mathcal{S}$ contains as many as $O(m/k)$ clusters. By Lemma 3.1, all the clusters in $\mathcal{S}$ as well as their $\mathcal{X}^i_\mathcal{C}$, $\mathcal{Y}^j_\mathcal{C}$, and $\mathcal{Z}^{h,t}_\mathcal{C}$ neighbors can be found in $O(m^2/k^2)$ time. We run the algorithm of Lemma 3.8 on all the clusters

in $\mathcal{S}$. If either $\mathcal{S} = \emptyset$ or there is no bridge in each cluster in $\mathcal{S}$ separating x and y, we return *true*. Otherwise, there is at least one bridge internal to a cluster and we return *false*.

LEMMA 4.1. *A Same2EdgeBlock(x,y) correctly returns* true *if the two vertices x and y are in the same 2-edge-connected component of G, and returns* false *otherwise. The total time required is $O(k + (m^2/k^2))$. When the graph is planar, the total time reduces to $O(k + (n/k)\log k)$.*

Proof. If there is a bridge separating x and y in $\mathcal{G}_{x,y}(G)$, then *Same2EdgeBlock(x,y)* returns *false*. This is correct because of Lemma 2.3 and requires $O(k + (m^2/k^2))$ time because of Lemma 2.2. Otherwise, we compute $\mathcal{S}$ in $O(m^2/k^2)$ time as described in Lemma 3.1 and run the algorithm given in Lemma 3.8 on all the clusters in $\mathcal{S}$. This gives a correct answer because of Lemmas 2.3 and 3.8. The total time required to test whether there is a bridge internal to a cluster in $\mathcal{S}$ is

$$\sum_{\mathcal{C} \in \mathcal{S}} d(\mathcal{C}) \log k$$

by Lemma 3.8. To bound this time, we notice that by Lemma 2.3 all the clusters in $\mathcal{S}$ are articulation points in $\mathcal{G}_{x,y}(G)$. Therefore, each cluster $\mathcal{C}'$ can be in each of the sets $\{\mathcal{X}^i_{\mathcal{C}}\}_{1 \le i \le p(\mathcal{C})}$, $\{\mathcal{Y}^j_{\mathcal{C}}\}_{1 \le j \le q(\mathcal{C})}$, and $\{\mathcal{Z}^{h,t}_{\mathcal{C}}\}_{3 \le h \le s(\mathcal{C}), 1 \le t \le r_h(\mathcal{C})}$ at most once for all possible choices of the articulation point $\mathcal{C}$ in $\mathcal{S}$ (that is, during the whole execution of the *Same2EdgeBlock(x,y)* operation). As a result, we have that

$$\sum_{\mathcal{C} \in \mathcal{S}} d(\mathcal{C}) = O\left(\frac{m}{k}\right).$$

Hence, the total time spent in checking whether there is a bridge inside the articulation points of $\mathcal{G}_{x,y}(G)$ is at most $O((m/k)\log k)$. This gives a total time of $O(k + (m^2/k^2) + (m/k)\log k) = O(k + (m^2/k^2))$ for each *Same2EdgeBlock* operation.

If the underlying graph G is planar and insertions of new edges leave it planar, we have that $m = O(n)$. Therefore, the topological partition contains $O(n/k)$ clusters. Since G is planar and contraction preserves planarity, the graph obtained by contracting each cluster of G into one vertex is itself planar. As a result, denoting by $d(\mathcal{C})$ the external degree of cluster $\mathcal{C}$, we have

$$\sum_{\mathcal{C}} d(\mathcal{C}) = O\left(\frac{n}{k}\right).$$

Then $\mathcal{G}_{x,y}(G)$ has $O(k + (n/k))$ vertices and edges. Therefore, each *Same2EdgeBlock* operation can now be performed in $O(k + (n/k) + (n/k)\log k) = O(k + (n/k)\log k)$ time. □

We now show how to support *InsertEdge(x,y)*. As usual, we denote by $\mathcal{C}_x$ and $\mathcal{C}_y$, the clusters containing x and y, respectively. When the new edge (x, y) is inserted, the degrees of x and y in the original graph increase by one. If either the degree of x or the degree of y becomes four, then the transformation given at the beginning of §2 has to be applied. If either degree becomes larger than four, then the transformation has been already applied but now must be updated. In both cases, this introduces at most a constant number of extra vertices and edges and can be easily handled. However, because of the insertion of extra vertices to keep a degree no greater than three, it may happen that the size of either $\mathcal{C}_x$ or $\mathcal{C}_y$ has now become greater than $3k - 2$. When the size of a cluster $\mathcal{C}$ becomes greater than $3k - 2$, we split $\mathcal{C}$ into two clusters $\mathcal{C}'$ and $\mathcal{C}''$.

LEMMA 4.2. *A cluster $\mathcal{C}$ whose size is greater than $3k-2$ but is still smaller than $4k$ can be split into two clusters, $\mathcal{C}'$ and $\mathcal{C}''$, such that they are connected and of size at least k and at most $3k-2$. The total time required to update all the data structures because of this splitting is $O(k)$.*

Proof. Since $\mathcal{C}$ has size $O(k)$, two clusters, $\mathcal{C}'$ and $\mathcal{C}''$, that satisfy the cluster requirements, i.e., they are connected and of size between k and $3k-2$, can be found in $O(k)$ time (see, for instance, [20]). We now analyze the time needed to update all the data structures because of this splitting. The changes involve the data structures pertinent to the newly created clusters, $\mathcal{C}'$ and $\mathcal{C}''$, all the balanced search trees $E(\mathcal{C}',\mathcal{C}_i)$ and $E(\mathcal{C}'',\mathcal{C}_j)$ (for any clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ adjacent, respectively, to $\mathcal{C}'$ and $\mathcal{C}''$), and the super-graph $\widetilde{G}$.

The full and tree representations of the new clusters $\mathcal{C}'$ and $\mathcal{C}''$ as well as the balanced search trees pertinent to $\mathcal{C}'$ and $\mathcal{C}''$ (i.e., $E(\mathcal{C}_i,\mathcal{C}')$ and $E(\mathcal{C}_j,\mathcal{C}'')$ for any clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ adjacent to $\mathcal{C}'$ and $\mathcal{C}''$, respectively) can be recomputed from scratch in $O(k)$, as explained in Lemma 2.4. The other update needed is to compute the balanced search trees $E(\mathcal{C}',\mathcal{C}_i)$ and $E(\mathcal{C}'',\mathcal{C}_j)$ for any clusters $\mathcal{C}_i$ and $\mathcal{C}_j$ that are adjacent to $\mathcal{C}'$ and $\mathcal{C}''$. We do this by visiting the old balanced search trees $E(\mathcal{C},\mathcal{C}_h)$ according to increasing preorder numbers. Each visited item is inserted either in $E(\mathcal{C}',\mathcal{C}_h)$ or in $E(\mathcal{C}'',\mathcal{C}_h)$, depending on whether the corresponding edge is incident now to $\mathcal{C}'$ or $\mathcal{C}''$. Because of the preorder numbering, items are always inserted at the end of $E(\mathcal{C}',\mathcal{C}_h)$ and $E(\mathcal{C}'',\mathcal{C}_h)$. As a result, these two balanced search trees can be computed in a total of $O(|E(\mathcal{C},\mathcal{C}_h)|)$ time. The total time for computing all these balanced search trees is, therefore, still $O(k)$. Because of the splitting, the super-graph $\widetilde{G}$ must also be updated. But this can be easily performed within the $O(k)$ time bound. Consequently, all the updates needed can be carried out in a total of $O(k)$ time. □

When the edge (x,y) has to be inserted, we apply possibly the transformation to keep vertices of degree no more than three, and split either cluster $\mathcal{C}_x$ or cluster $\mathcal{C}_y$ if its size exceeds $3k-2$. As explained above, this can be done in $O(k)$. Now let $\mathcal{C}_x$ and $\mathcal{C}_y$ denote the clusters containing x and y after the splitting has been performed. In order to perform the actual insertion of edge (x,y), we distinguish two cases depending on whether (x,y) is an edge internal to a cluster or an inter-cluster edge.

If $\mathcal{C}_x = \mathcal{C}_y = \mathcal{C}$, then (x,y) is an internal edge. As a result, the insertion of (x,y) changes the interior of $\mathcal{C}$, that is, the full and the tree representation $\mathcal{F}(\mathcal{C})$ and $\mathcal{T}(\mathcal{C})$ of $\mathcal{C}$. The change in $\mathcal{T}(\mathcal{C})$ may induce changes in the balanced search tree $E(\mathcal{C}',\mathcal{C})$ for each cluster $\mathcal{C}'$ neighbor of $\mathcal{C}$. But this implies that only the data structures pertinent to $\mathcal{C}$ need to be updated. By Lemma 2.4, all these data structures can be recomputed from scratch in $O(k)$ time.

If $\mathcal{C}_x \neq \mathcal{C}_y$, then we have to insert an inter-cluster edge (x,y). The updates needed are in the full representations of clusters, $\mathcal{C}_x$ and $\mathcal{C}_y$, and in the two balanced search trees $E(\mathcal{C}_x,\mathcal{C}_y)$ and $E(\mathcal{C}_y,\mathcal{C}_x)$. As said before, this can be done in $O(k)$ time by simply reinitializing the data structures pertinent to clusters $\mathcal{C}_x$ and $\mathcal{C}_y$. Furthermore, we may have to update the edges between $\mathcal{C}_x$ and $\mathcal{C}_y$ in $\widetilde{G}$, but this can be easily done within the $O(k)$ time bound.

In summary, inserting an edge (x,y) causes at most two clusters to be split and a constant number of clusters to be updated. By Lemma 4.2, each cluster can be split in $O(k)$ time, and by Lemma 2.4 each cluster and its pertinent data structures can be recomputed from scratch in $O(k)$ each time an update is needed. This leads to the following lemma.

LEMMA 4.3. *The total time required to update our data structure because of an Insert-Edge operation is $O(k)$.*

The deletion of an edge (x, y) can be performed as follows. Again we denote by $\mathcal{C}_x$ and $\mathcal{C}_y$ the clusters containing, respectively, x and y. When the edge (x, y) is being deleted, we may have to apply the reverse of the transformation used to keep the vertex degrees no greater than three. This can be easily taken care of, since it deletes only a constant number of vertices and edges and it involves only a constant number of clusters. If because of this either $\mathcal{C}_x$ or $\mathcal{C}_y$ now has size less than k, then it is merged with an adjacent cluster. If this adjacent cluster now has size greater than $3k - 2$, it must be split into two clusters. Since the cluster resulting from this merging has size at most $(k-1) + (3k-2) < 4k$, this splitting can be carried out in $O(k)$ time because of Lemma 4.2. Notice that there is no further propagation of merging and splitting.

Merging two clusters involves updates similar to splitting a cluster and can be carried out in $O(k)$ time as the following lemma shows.

LEMMA 4.4. *Let $\mathcal{C}'$ be a cluster whose size is less than k and $\mathcal{C}''$ be a cluster adjacent to $\mathcal{C}'$. Then $\mathcal{C}'$ and $\mathcal{C}''$ can be merged into a new cluster $\mathcal{C}$ in $O(k)$ time.*

Proof. The updates needed are in the data structures pertinent to the newly created cluster $\mathcal{C}$, in all the balanced search trees $E(\mathcal{C}, \mathcal{C}_i)$ for any cluster $\mathcal{C}_i$ adjacent to $\mathcal{C}$, and in the super-graph $\widetilde{G}$.

All the data structures pertinent to the new cluster $\mathcal{C}$ can be recomputed from scratch in $O(k)$ by Lemma 2.4. The other update needed is to compute the balanced search trees $E(\mathcal{C}, \mathcal{C}_i)$ for any cluster $\mathcal{C}_i$ adjacent to $\mathcal{C}$. We compute $E(\mathcal{C}, \mathcal{C}_i)$ by merging the two old balanced search trees $E(\mathcal{C}', \mathcal{C}_i)$ and $E(\mathcal{C}'', \mathcal{C}_i)$, inserting one item at a time into $E(\mathcal{C}, \mathcal{C}_i)$ according to increasing preorder numbers. Because of this, items will always be inserted at the end of $E(\mathcal{C}, \mathcal{C}_i)$, and, therefore, this balanced search tree can be computed in a total of $O(|E(\mathcal{C}, \mathcal{C}_i)|)$ time. Since we can have at most $O(k)$ edges incident to cluster $\mathcal{C}$, the total time required to compute all the balanced search trees $E(\mathcal{C}, \mathcal{C}_i)$ for clusters $\mathcal{C}_i$ adjacent to $\mathcal{C}$ is $O(k)$. Finally, because of the clusters merging, we may have to merge two super-vertices of $\widetilde{G}$ into one. But this can be easily performed within the $O(k)$ time bound. □

By Lemmas 4.2 and 4.4, the total time required to determine whatever splits and merges are needed as well as to perform them along with the reorganization of the data structures involved is $O(k)$. Let $\mathcal{C}_x$ and $\mathcal{C}_y$ denote the two clusters containing x and y after all these possible splits and merges have been performed.

If $\mathcal{C}_x \neq \mathcal{C}_y$, deleting edge (x, y) requires recomputing the full representations of the two clusters and updating the balanced search trees, $E(\mathcal{C}_x, \mathcal{C}_y)$ and $E(\mathcal{C}_y, \mathcal{C}_x)$. Furthermore, we may have to update the edges between $\mathcal{C}_x$ and $\mathcal{C}_y$ in $\widetilde{G}$. As explained before, all this can be done in a total of $O(k)$ time.

If $\mathcal{C}_x = \mathcal{C}_y = \mathcal{C}$, then we have the following two cases. If the deletion of (x, y) leaves the cluster $\mathcal{C}$ connected, then we need to recompute the full and tree representation of $\mathcal{C}$, as well as the balanced search trees $E(\mathcal{C}', \mathcal{C})$ for each $\mathcal{C}'$ neighbor of $\mathcal{C}$. Once again, this requires $O(k)$ time by Lemma 2.4. Otherwise, if the deletion of edge (x, y) disconnects $\mathcal{C}$, it can no longer be a cluster. Denote by $\mathcal{C}'$ and $\mathcal{C}''$ the two connected components into which the deletion of (x, y) splits cluster $\mathcal{C}$. Since $\mathcal{C}$ has size $O(k)$, this splitting may be performed in a total of $O(k)$ time by Lemma 4.2. If either $\mathcal{C}'$ or $\mathcal{C}''$ has size less than k, then we merge it with an adjacent cluster. If this adjacent cluster now has more than $3k - 2$ vertices, it must be split into two clusters. Note that no further merges and splits are needed. By Lemmas 4.2 and 4.4, the total time required to determine whatever splits and merges are needed as well as to perform them along with both the reorganization of the data structures involved and the updates needed in $\widetilde{G}$ is $O(k)$.

In summary, deleting an edge (x, y) causes a constant number of cluster splits and merges, and at most a constant number of cluster updates. By Lemmas 2.4, 4.2, and 4.4 all this can be done in $O(k)$ time. Therefore, we have the following lemma.

LEMMA 4.5. *The total time required to update our data structure because of a DeleteEdge operation is $O(k)$.*

5. Time complexity. Our implementation of the *Same2EdgeBlock*, *InsertEdge*, and *DeleteEdge* operations achieve the following bounds.

THEOREM 5.1. *The data structure above supports each InsertEdge, DeleteEdge, and Same2EdgeBlock operation in $O(m^{2/3})$ amortized time on a connected graph, where m is the current number of edges in the graph. For planar connected graphs, this time reduces to $O(\sqrt{n \log n})$. The total space required by the data structure is $O(m)$.*

Proof. By Lemmas 4.1, 4.3, and 4.5, each *InsertEdge* and *DeleteEdge* operation can be performed in $O(k)$ time, while *Same2EdgeBlock* requires $O(k + (m^2/k^2))$ time. Choosing $k = \lceil m^{2/3} \rceil$ gives the claimed bounds. However, this choice of k depends on the number m of current edges, which may change due to *InsertEdge* and *DeleteEdge* operations. We circumvent this by recomputing the topological partition of G every time m gets either twice larger or twice smaller than before. Since recomputing the topological partition, as well as the full and the tree representations, and the balanced search trees of the new clusters can be done in linear time, this can be amortized against the number of edges inserted and deleted after the last time we recomputed the topological partition. As a result, the times for *InsertEdge*, *DeleteEdge*, and *Same2EdgeBlock* become amortized.

If the underlying graph G is planar and insertions of new edges leave it planar, the time for *Same2EdgeBlock* reduces to $O(k+(n/k)\log k)$ as shown in Lemma 4.1. Choosing $k = \lceil \sqrt{n \log n} \rceil$ gives the $O(\sqrt{n \log n})$ amortized bound for planar graphs.

The total space complexity of our data structure can be analyzed as follows. Throughout the sequence of operations, we maintain the super-graph $\widetilde{G}$, which is of size $O(m^2/k^2)$ $= O(m^{2/3})$. Furthermore, by Lemma 2.4 each cluster $\mathcal{C}$ requires at most $O(k)$ space to store its pertinent data structures. Since there can be at most $O(m/k)$ clusters, the total space required by our data structure is $O(m)$. □

The bound for planar graphs can be further reduced, as the following theorem shows.

THEOREM 5.2. *Each InsertEdge, DeleteEdge, and SameEdgeBlock operation on planar graphs can be supported in $O(\sqrt{n \log\log n})$ amortized time, provided that InsertEdge operations leave the graph planar. The space required is $O(n)$.*

Proof. The bottleneck of our algorithm for planar graphs is how to check whether there is a bridge inside a cluster. Indeed, we accomplished this task in a total of $O((n/k)\log k)$. The more time consuming tasks were performing range queries in the balanced search trees $E(\mathcal{C}_i, \mathcal{C}_j)$ and sorting the vertices $\gamma'_1, \gamma'_2, \dots, \gamma'_r$ (as shown in Lemma 3.8). Notice that for each pair of clusters $\mathcal{C}_i$ and $\mathcal{C}_j$, the set $E(\mathcal{C}_i, \mathcal{C}_j)$ contains preorder numbers in $\mathcal{T}(\mathcal{C}_j)$, i.e., integers in the range $[1, 3k - 2]$. Similarly, the vertices $\gamma'_1, \gamma'_2, \dots, \gamma'_r$ have to be sorted according to their preorder numbers, again integers in the range $[1, 3k - 2]$. We exploit this by using Johnson's stratified trees [33] instead of balanced search trees to represent the sets $E(\mathcal{C}_i, \mathcal{C}_j)$. Each range query can now be performed more efficiently in $O(\log\log k)$ time, at the expense of more space usage that increases from $O(|E(\mathcal{C}_i, \mathcal{C}_j)|)$ to $O(k)$. The initialization of Johnson's stratified tree containing $O(k)$ items can be still accomplished in $O(k)$ time [37], which implies that the time required to initialize the data structures pertinent to a cluster (as analyzed in Lemma 2.4) is still $O(k)$. Furthermore, the vertices $\gamma'_1, \gamma'_2, \dots, \gamma'_r$ can be sorted more efficiently in $O(d(\mathcal{C}) \log\log k)$ time at the expense of $O(k)$ space. Consequently, us-

ing stratified trees leads to a more efficient implementation of *Same2EdgeBlock* in time $O(k + (n/k) + (n/k)\log\log k) = O(k + (n/k)\log\log k)$. Following exactly the same argument given earlier, it can be shown that during either an *InsertEdge* or a *DeleteEdge* operation the time required to update a cluster, or to merge two clusters into a new cluster, or to split a cluster into two is still $O(k)$. As a result, the time required to perform *InsertEdge* and *DeleteEdge* operations is again $O(k)$. Choosing $k = \lceil \sqrt{n \log\log n}\, \rceil$, gives an $O(\sqrt{n \log\log n}\,)$ time bound per operation in case of planar graphs.

The total space required by this data structure is $O(n)$ (as shown in Theorem 5.1), plus the total space required by all the stratified trees. We now bound the total number of stratified trees needed. Consider the super-graph $\widetilde{G}$ obtained (i) by contracting each cluster into a super-vertex and (ii) by reducing to two the number of multiple edges between any two super-vertices. Notice that the number of edges in $\widetilde{G}$ is an upper bound on the total number of stratified trees needed. But since the original graph G is planar, and contraction preserves planarity, $\widetilde{G}$ is itself planar. Because $\widetilde{G}$ has $O(n/k)$ vertices, it has at most $O(n/k)$ edges. Therefore, we have at most $O(n/k)$ stratified trees in our data structure, each of which requires $O(k)$ space. Consequently, the extra space needed is $O((n/k)k) = O(n)$. □

Whenever needed, the times given in Theorems 5.1 and 5.2 can be made worst-case as the following theorem shows.

THEOREM 5.3. *The data structure above supports each InsertEdge, DeleteEdge, and Same2EdgeBlock operation in $O(m^{2/3})$ time in the worst case, where m is the current number of edges in the connected graph. For planar connected graphs, this time reduces to $O(\sqrt{n \log\log n}\,)$.*

Proof. Let m_t be the number of edges in the graph at time t. We claim that an update at time t for general graphs can be carried out in $O(m_t^{2/3})$ worst-case time. This can be achieved as follows. Let $k_t = \lceil m_t^{2/3} \rceil$. When the value of k changes due to an *InsertEdge* or *DeleteEdge* operation, there will be at least $\frac{1}{2}\lceil m_t^{2/3} \rceil$ more updates before k becomes twice as large or twice as small as they were before. The idea is to adjust a constant number of clusters each time there is an update. This sums up to a total of $O(m_t^{2/3})$ cluster adjustments. Since there will be no more than

$$\frac{m_t}{k_t} \le \left\lceil m_t^{1/3} \right\rceil \le O(m_t^{2/3})$$

clusters that need to be adjusted, the adjustments may be accomplished before a new round of adjustments is initiated. Thus every time an insertion occurs, the clusters can be scanned to find any cluster that is too small, and a constant number of these clusters can be combined with a neighbor if needed. Similar operations are performed during an edge deletion. Clearly, the same argument applies to planar graphs. □

6. Dealing with unconnected graphs. We end this section by showing how to deal with unconnected graphs within the same time bounds. Let G be an unconnected graph. If G consists of q connected components, we augment G by inserting $q - 1$ *dummy* edges that make it connected, such that there is no cycle containing two dummy edges. This does not change the 2-edge-connected components of G, and two vertices x and y are in the same 2-edge-connected component in the augmented graph if and only if they are in the same 2-edge-connected component in the original graph. Furthermore, we assign cost 1 to each edge of G and cost 2 to each dummy edge, and we maintain a topological partition of a minimum spanning tree of the augmented graph.

To insert an edge $e = (x, y)$, we first check whether there is already a dummy edge between x and y. If so, we decrease its cost from 2 to 1. Otherwise, we insert the

edge (x, y) of cost 1 and update the minimum spanning tree in $O(\sqrt{m})$ time using the algorithm by Frederickson [20]. If (x, y) enters the minimum spanning tree T and a dummy edge e' leaves T, we delete e'. This can be accomplished within the $O(m^{2/3})$ and $O(\sqrt{n \log \log n})$ time bounds.

To delete an edge $e = (x, y)$, we first check whether e is in the minimum spanning tree T. If e is not in T, we simply delete e. If e is in T, we check, by using the algorithm by Frederickson [20], whether there is a replacement edge for e. If there is such a replacement, we simply delete e. Otherwise, removing e will disconnect the augmented graph and therefore we make e dummy by increasing its cost from 1 to 2. Once again, this does not change the $O(m^{2/3})$ and $O(\sqrt{n \log \log n})$ time bounds. This gives us the following theorem.

THEOREM 6.1. *The data structure above supports each InsertEdge, DeleteEdge, and Same2EdgeBlock operation in $O(m^{2/3})$ time in the worst case on a general graph, where m is the current number of edges. For planar graphs, this time reduces to $O(\sqrt{n \log \log n})$.*

7. Concluding remarks. In this paper we have described efficient algorithms to maintain the 2-edge-connected components of an undirected graph under insertions and deletions of edges. We have shown how to support each operation in $O(m^{2/3})$ for general graphs, and in $O(\sqrt{n \log \log n})$ for planar graphs (we allow changes in the embedding). Frederickson [21] has recently improved these bounds to $O(\sqrt{m})$, and to $O(\log^3 n)$ for embedded planar graphs. Very recently, we sped up Frederickson's result and now have an $O(\sqrt{n}\, \log(m/n))$ algorithm for fully dynamic 2-edge connectivity [16].

We can also achieve the same $O(m^{2/3})$ bound per operation for the fully dynamic maintenance of the 3-edge-connected components of a graph [22].

The problem of maintaining the biconnected components of a graph during edge insertions and deletions deserves further study. We have been able to prove that if the graph is planar, this problem can be solved in $O(n^{2/3})$ worst-case time per operation [23], and we are currently exploring the case of general graphs.

Acknowledgments. We are grateful to Greg Frederickson for his valuable comments. We also would like to thank Dany Breslauer, Kurt Mehlhorn, Neil Sarnak, and Moti Yung for useful discussions.

REFERENCES

[1] R. AGRAWAL, A. BORGIDA, AND H. V. JAGADISH, *Efficient management of transitive relationships in large data and knowledge bases*, in Proceedings of the ACM-SIGMOD International Conference on Management of Data, 1989.

[2] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] B. ALPERN, R. HOOVER, B. ROSEN, P. SWEENEY, AND F. K. ZADECK, *Incremental evaluation of computational circuits*, in Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 32–42.

[4] G. AUSIELLO AND G. F. ITALIANO, *On-line algorithms for polynomially solvable satisfiability problems*, J. Logic Programming, 10 (1991), pp. 69–90.

[5] G. AUSIELLO, G. F. ITALIANO, A. MARCHETTI-SPACCAMELA, AND U. NANNI, *Incremental algorithms for minimal length paths*, J. Algorithms, 12 (1991), pp. 615–638.

[6] G. AUSIELLO, A. MARCHETTI-SPACCAMELA, AND U. NANNI, *Dynamic maintenance of paths and path expressions in graphs*, in Proceedings of the International Symposium on Symbolic and Algebraic Computation, Lecture Notes in Computer Science 358, Springer-Verlag, Berlin, 1989, pp. 1–12.

[7] B. AWERBUCH AND Y. SHILOACH, *New connectivity and* MSF *algorithms for shuffle-exchange networks and* PRAM, IEEE Trans. Computers, C-36 (1987), pp. 1258–1263.

[8] M. R. BROWN AND R. E. TARJAN, *Design and analysis of a data structure for representing sorted lists*, SIAM J. Comput., 9 (1980), pp. 594–614.

[9] A. L. BUCHSBAUM, P. C. KANELLAKIS, AND J. S. VITTER, *A data structure for arc insertion and regular path finding*, in Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 22–31.

[10] M. BURKE AND B. G. RYDER, *Incremental iterative data flow analysis algorithms*, Tech. Report LCSR-TR-96, Dept. of Computer Science, Rutgers University, New Brunswick, NJ, 1987.

[11] M. D. CARROL AND B. G. RYDER, *Incremental data flow analysis via dominator and attribute updates*, in Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1988, pp. 274–284.

[12] F. CHIN AND D. HOUK, *Algorithms for updating minimum spanning trees*, J. Comput. Syst. Sci., 16 (1978), pp. 333–344.

[13] R. F. COHEN AND R. TAMASSIA, *Dynamic expression trees and their applications*, in Proceedings of the 2nd Annual ACM-SIAM Symposium on Discrete Algorithms, 1991, pp. 52–61.

[14] G. DI BATTISTA AND R. TAMASSIA, *Incremental planarity testing*, in Proceedings of the 30th Annual Symposium on Foundations of Computer Science, 1989, pp. 436–441.

[15] ———, *On-line graph algorithms with* SPQR-*trees*, in Proceedings of the 17th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, 443, Springer-Verlag, Berlin, 1990, pp. 598–611.

[16] D. EPPSTEIN, Z. GALIL, G. F. ITALIANO, AND A. NISSENZWEIG, *Sparsification – A technique for speeding up dynamic graph algorithms*, manuscript, 1992.

[17] D. EPPSTEIN, G. F. ITALIANO, R. TAMASSIA, R. E. TARJAN, J. WESTBROOK, AND M. YUNG, *Maintenance of a minimum spanning forest in a dynamic planar graph*, in Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 1–11; J. Algorithms, to appear.

[18] S. EVEN AND H. GAZIT, *Updating distances in dynamic graphs*, Methods Oper. Res., 49 (1985), pp. 371–387.

[19] S. EVEN AND Y. SHILOACH, *An on-line edge deletion problem*, J. Assoc. Comput. Mach., 28 (1981), pp. 1–4.

[20] G. N. FREDERICKSON, *Data structures for on-line updating of minimum spanning trees*, SIAM J. Comput., 14 (1985), pp. 781–798.

[21] ———, *Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees*, in Proceedings of the 32nd Annual Symposium on Foundations of Computer Science, 1991, pp. 632–641.

[22] Z. GALIL AND G. F. ITALIANO, *Fully dynamic algorithms for 3-edge-connectivity*, Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, 1991.

[23] ———, *Maintaining biconnected components of dynamic planar graphs*, in Proceedings of the 18th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 510, Springer-Verlag, Berlin, 1991, pp. 339–350.

[24] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[25] D. HAREL AND R. E. TARJAN, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[26] R. HOOVER, *Incremental graph evaluation*, Ph.D. thesis, Tech. Report 87-836, Dept. of Computer Science, Cornell University, Ithaca, NY, 1987.

[27] N. HORSPOOL, *Incremental generation of* LR *parsers*, Tech. Report, Dept. of Computer Science, University of Victoria, Victoria, BC, Canada, 1988.

[28] T. IBARAKI AND N. KATOH, *On-line computation of transitive closure for graphs*, Inform. Process. Lett., 16 (1983), pp. 95–97.

[29] G. F. ITALIANO, *Amortized efficiency of a path retrieval data structure*, Theoret. Comput. Sci., 48 (1986), pp. 273–281.

[30] ———, *Finding paths and deleting edges in directed acyclic graphs*, Inform. Process. Lett., 28 (1988), pp. 5–11.

[31] G. F. ITALIANO, A. MARCHETTI-SPACCAMELA, AND U. NANNI, *Dynamic data structures for series parallel digraphs*, in Proceedings on the Workshop on Algorithms and Data Structures, Lecture Notes in Computer Science 382, Springer-Verlag, Berlin, 1989, pp. 352–372.

[32] H. V. JAGADISH, *A compression technique to materialize transitive closure*, ACM Trans. Database Systems, 15 (1990), pp. 558–598.

[33] D. B. JOHNSON, *A priority queue in which initialization and queue operations take* $O(\log \log D)$ *time*, Math. Systems Theory, 15 (1982), pp. 295–309.

[34] S. KAPLAN, *Incremental attribute evaluation on graphs*, Tech. Report UIUC-DCS-86-18309, University of Illinois at Urbana-Champaign, Urbana, IL, 1986.

[35] J. A. LaPoutré and J. van Leeuwen, *Maintenance of transitive closure and transitive reduction of graphs*, in Proceedings of the Workshop on Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science, 314, Springer-Verlag, Berlin, 1988, pp. 106–120.

[36] C. C. Lin and R. C. Chang, *On the dynamic shortest path problem*, in Proceedings of the International Workshop on Discrete Algorithms and Complexity, 1989, pp. 203–212.

[37] K. Mehlhorn, personal communication, 1990.

[38] K. Mehlhorn, S. Näher, and H. Alt, *A lower bound for the complexity of the union-split-find problem*, SIAM J. Comput., 17 (1988), pp. 1093–1102.

[39] J. H. Reif, *A topological approach to dynamic graph connectivity*, Inform. Process. Lett., 25 (1987), pp. 65–70.

[40] H. Rohnert, *A dynamization of the all pairs least cost path problem*, in Proceedings of the 2nd Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science, 182, Springer-Verlag, Berlin, 1985, pp. 279–286.

[41] B. Schieber and U. Vishkin, *On finding lowest common ancestors: simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.

[42] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, J. Comput. Syst. Sci., 24 (1983), pp. 362–381.

[43] ———, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652–686.

[44] P. M. Spira and A. Pan, *On finding and updating spanning trees and shortest paths*, SIAM J. Comput., 4 (1975), pp. 375–380.

[45] R. Tamassia, *A dynamic data structure for planar graph embedding*, in Proceedings of the 15th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science, 317, Springer-Verlag, Berlin, 1988, pp. 576–590.

[46] R. Tamassia and F. P. Preparata, *Dynamic maintenance of planar digraphs, with applications*, Algorithmica, 5 (1990), pp. 509–527.

[47] R. E. Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

[48] ———, *A class of algorithms which require nonlinear time to maintain disjoint sets*, J. Comput. Syst. Sci., 18 (1979), pp. 110–127.

[49] R. E. Tarjan and J. van Leeuwen, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245–281.

[50] R. E. Tarjan and U. Vishkin, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput., 14 (1985), pp. 862–864.

[51] J. Westbrook, *Algorithms and data structures for dynamic graph problems*, Ph.D. thesis, Tech. Report CS-TR-229-89, Dept. of Computer Science, Princeton University, Princeton, NJ, October 1989.

[52] J. Westbrook and R. E. Tarjan, *Maintaining bridge-connected and biconnected components on-line*, Tech. Report CS-TR-228-89, Dept. of Computer Science, Princeton University, Princeton, NJ, August 1989; Algorithmica, to appear.

[53] M. Yannakakis, *Graph theoretic methods in database theory*, in Proceedings of the ACM Conference on Principles of Database Systems, 1990, pp. 230–242.

[54] A. C. Yao, *Should tables be sorted?*, J. Assoc. Comput. Mach., 31 (1984), pp. 245–281.

[55] D. M. Yellin, *A dynamic transitive closure algorithm*, Res. Report 13535, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY, 1988.

[56] D. M. Yellin and R. Strom, INC: *a language for incremental computations*, in Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation, 1988, pp. 115–124.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1070–1099, December 1992

© 1992 Society for Industrial and Applied Mathematics
005

WORK-OPTIMAL ASYNCHRONOUS ALGORITHMS FOR SHARED MEMORY PARALLEL COMPUTERS*

CHARLES MARTEL†, ARVIN PARK†, AND RAMESH SUBRAMONIAN‡

Abstract. This paper develops shared memory algorithms for asynchronous processor systems that require the same expected work as the best PRAM algorithms. These algorithms operate efficiently under general asynchronous processor behavior (where individual processor speeds are allowed to vary widely over time). This paper achieves these results by employing a methodology that uses randomization to schedule subtasks of a parallel program. The resulting algorithms allow processors to (i) have arbitrary asynchronous behavior, (ii) have fail-stop errors, (iii) join a computation at any time, and (iv) have no unique identifiers.

This paper develops a performance metric for asynchronous parallel computations, called *work*, which is the total number of instructions (including busy-waiting instructions) performed by a collection of parallel processors during a computation. The main result is to compute any associative function of n variables with $O(n)$ expected work, using up to $n/\log n \log^* n$ asynchronous processors, and with $O(n \log n)$ expected work using up to n processors. These results provide a synchronization primitive that can be used to transform any PRAM program into an asynchronous PRAM program.

Key words. PRAM, parallel computation, synchronization, fault-tolerance, randomized algorithm shared memory

AMS(MOS) subject classifications. primary 68Q22; secondary 68Q25, 68R05

1. Introduction. This paper addresses a problematic assumption that is implicit in the commonly used PRAM model of shared memory parallel computation. Processors are assumed to operate synchronously in the PRAM model. However, maintaining this synchrony can incur severe performance penalties in practical large scale parallel systems. Hence, a number of recent papers have considered relaxing the PRAM assumption of synchrony [CZ89], [CZ90b], [Gib89], [KS89], [KPS90], [KRS90], [MPS89], [MSP90], [MS90], [SM90].

Variations in processor execution speed can arise from a number of sources such as clock skew, varying processor load, multiprogramming, interrupts, page faults, cache misses, and differences in processor speeds. These variations make it impractical to enforce synchrony for a large number of parallel processors between each parallel step. This motivates us to investigate parallel algorithms that operate efficiently in an asynchronous environment.

In an asynchronous environment where processor speeds can vary widely and arbitrarily, running time is not an appropriate measure of performance. We measure the performance of our algorithms in terms of *work*, which is the total number of instructions executed by all processors during the execution of the program. Since this includes all busy waiting instructions, work is simply a generalization of the processor-time product. In an asynchronous system, it is possible for faster processors to contribute a great deal to the work, while slower processors contribute very little. If a processor working at 10 instructions per time unit must wait three time units for an intermediate result, we charge those 30 potential instructions as part of the computation's work. If the processors are "close" to being synchronous (as is the case in a number of other asynchronous

*Received by the editors December 5, 1990; accepted for publication (in revised form) November 7, 1991. This research was supported by the Computer Science Division of the University of California at Davis and National Science Foundation grants CCR 87-22848, CCR 90-11280, and CCR-90-23727 and University of California MICRO grants 89-165 and 91-118.

†Division of Computer Science, University of California, Davis, California 95616 (martel@cs.ucdavis.edu and park@cs.ucdavis.edu).

‡Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720 (subramon@harmony.berkeley.edu).

models), the running time of an n processor algorithm will be $O(work/n)$. The relation between work and time is described formally in §5.2.

In this paper we show that for many problems, asynchronous computations can be performed using no more asymptotic expected work than for computations where processors run in lock-step synchrony. We achieve these strong results with a methodology that uses randomization to perform processor assignment in asynchronous parallel shared memory computations. This methodology produces algorithms that operate efficiently even if processor speeds vary greatly and unpredictably over time. (If a processor is very slow, it will eventually be overtaken by a faster processor that gets randomly assigned to the same task.) Our use of randomization for processor assignment differs from any results we are familiar with in this area.

Randomization also leads to algorithms that are resistant to processor failures and produces algorithms that can accommodate a variable number of processors. As the number of processors involved in a computation grows, the probability of processor failure also increases. For this reason it is desirable to produce parallel algorithms that are resistant to processor failures. If a parallel algorithm can accommodate a variable number of parallel processors over time, processors can join the execution of an algorithm as they become freed from other tasks or leave as they are needed elsewhere. This improves system utilization because processors are not required to wait until a certain number of processors become available to begin execution of a task. (Of course there is a limit to the number of processors that can be efficiently utilized on a given problem instance.)

Much recent work has been done on models of asynchronous PRAMs. Kruskal, Rudolph, and Snir [KRS90] first proposed an asynchronous model where all interleavings of processors' instruction streams are possible. Cole and Zajicek [CZ89] proposed the APRAM, which is very similar to our model. They showed that summation and graph connectivity could be solved under arbitrary asynchronous behavior. In [CZ90b], they study asynchronous settings where in a parallel step each processor flips a biased coin. All processors that get heads do their next instruction in the current parallel step. In their unbounded delay model, the processors that get tails are idle in the step. In their bounded delay model, the processors that get tails each take k time units to complete their next instruction. They prove sharp expected time bounds for simple summation and pointer jumping algorithms. However, their algorithms perform very poorly in general asynchronous settings. By contrast, our algorithms for summation and pointer jumping are efficient in general asynchronous settings and are fast when analyzed in the [CZ90b] models. Nishimura proposed a model where asynchrony is modeled by the effect of different interleavings of instruction streams [Nis90]. She investigated tree and pointer jumping algorithms as well as more general algorithms and showed that they perform well when all interleavings are assumed to be equally likely. Gibbons considers a model that charges for synchronization costs and for the time to access global memory [Gib89]. He assumes an external synchronization primitive and suggests balancing the amount of work between synchronization steps with the cost of synchronization. The algorithms proposed in all of these papers are efficient when all processors run at similar speeds. However, they all perform very poorly under general asynchronous behavior.

An important motivation for developing an asynchronous model of parallel computation is the hope that it can provide a bridge from an ideal, abstract model to real machines. Valiant [Val90], Vishkin [Vis84], and others have argued for the importance of efficient transformations from PRAM programs to programs that run on more realistic models. The PRAM is an attractive choice for the abstract machine since it has proven to be an effective model for the design of efficient parallel algorithms. We believe

that it is particularly important to have automatic transformations from the PRAM to an asynchronous PRAM model because of the difficulty of writing correct asynchronous programs. It is much easier to prove a synchronous parallel program correct than an equivalent asynchronous program.

Several researchers have investigated transformations from PRAMs to asynchronous settings. Kanellakis and Shvartsman [KS89] suggested a general scheme for simulating most n-processor PRAM programs on an n-processor PRAM with stop-failures. They devised a clever but complicated deterministic scheme that simulates a single PRAM step using $O(n \log^2 n)$ work and up to n processors. Shvartsman [Shv89] improved this simulation to operate on any deterministic PRAM program, and he showed that the work is $O(n)$ per step using up to $n/\log^2 n$ processors. Independently, Kedem, Palem, and Spirakis [KPS90] also improved the [KS89] result to get simulations of all deterministic PRAM programs using $O(n \log\log n)$ expected work per step with up to $n/\log n$ processors on a PRAM with stop-failures. Because the Kanellakis and Shvartsman simulations are deterministic, their results hold even for adaptive adversaries that can determine processor failures knowing all future actions of the algorithm. The KPS simulation as well as the simulations presented in this paper use randomized algorithms that have good expected running times against oblivious adversaries that prespecify the processors' behavior.

In this paper, we develop a synchronization primitive that allows $n/\log n \log^* n$ processors to execute n PRAM instructions and then synchronize using $O(n)$ expected work. This synchronization primitive can be used to improve the [KPS90] simulation on PRAMs with stop-failures so that it uses $O(n)$ expected work per step. This synchronization primitive can also be used to simulate any PRAM program on a restricted asynchronous model with no loss in asymptotic efficiency [MSP90].

The simulations of [KS89], [KPS90], [MSP90] all place some limitations on the asynchronous behavior of the simulating processors. We show that computations with a very predictable structure can be simulated with no loss in efficiency on an asynchronous PRAM that allows arbitrary asynchronous behavior. In particular, we show that any function computable by a bounded-degree fan-in circuit of size S can be computed with $O(S)$ expected work on an asynchronous PRAM. Stockmeyer and Vishkin have shown that any PRAM computation can be transformed into a bounded degree fan-in circuit [SV84]. However, the resulting circuit can be quite large, and may consequently lead to a greatly increased running time for the transformed version of the computation.

Unpredictable asynchronous behavior can occur when processors fail, fix themselves, and then rejoin a computation. This type of behavior is common in a number of fault tolerant systems [Joh89]. Our algorithms work well in this setting. However, neither the [Shv89] nor the [KPS90] simulations can be used directly to simulate a system that allows processors to rejoin the computation or to simulate an asynchronous PRAM.

Researchers in distributed computation have also studied asynchronous processors that communicate via a shared memory [Her88], [ALS90], [AH90]. Using atomic reads and writes it is impossible to achieve two processor consensus and to implement wait-free versions of many standard data structures such as stacks and queues [Her88]. However, these impossibility results apply to settings where processors have private values. In our applications, we assume that all data is stored in global memory and is available to all processors. Thus the lower bounds for consensus and related problems with private values are not directly relevant to our setting.

In §2 of this paper we define a model for asynchronous parallel computation. In doing so we establish a metric for the expected "work" of parallel asynchronous algorithms.

Section 3 describes an algorithm to compute any associative function of n numbers with expected $O(n \log n)$ work using up to n processors. Section 4 describes a better algorithm, which computes any associative function of n numbers with expected $O(n)$ work using up to $n/ \log n \log^* n$ processors. Section 4.2 extends this result to hold for parallel prefix sums with the same bounds. Section 5 shows that the algorithm of §4 can be used as a synchronization primitive. We also discuss how to automatically convert PRAM programs to efficient asynchronous programs. Section 5.1 discusses methods to improve the worst case behavior of our algorithms and describes a class of settings in which our algorithms are fast as well as efficient. Section 6 shows that the results from §5 can be used to produce list ranking algorithms with expected $O(n \log n)$ work using up to $n/ \log n \log^* n$ asynchronous processors and expected $O(n \log n \log \log n)$ work using up to n processors. The same work bounds are also shown to apply for sorting. Section 7 contains conclusions.

2. Model. We now introduce the A-PRAM (Asynchronous Parallel Random Access Machine). The A-PRAM is similar to the ARBITRARY Concurrent Read Concurrent Write (CRCW) PRAM model (see [KR90], [KRS90] for excellent surveys), but allows asynchronous processors. We assume that the A-PRAM consists of a collection of unit cost RAMs [AHU74] that can perform atomic read and write operations to shared memory without contention. We also assume that each processor has access to an independent random number generator.

Our results hold for general asynchronous processor behavior, where instructions are completed at arbitrary real points in time. However, for ease of exposition, we present a model that is close to the standard CRCW PRAM model. In this model, time is divided into unit length slots in which a single low level RAM instruction (read, jump, write, ...) can be executed. However, we allow processors to have delays of an *arbitrary* number of slots between instructions. As a consequence of these delays, not all processors will perform an instruction in a given slot. In a slot, a processor can perform a single RAM instruction [AHU74], e.g., load, store, add.

Our model uses ARBITRARY concurrent writes; so if there are two or more writes to a single variable in a slot, then an arbitrary one succeeds. A write in a slot alters the value of the variable only after all the reads have occurred. A read in a slot returns the value of the variable at the end of the previous slot. More formally, for a variable X, the value returned by a read(X) in slot s is the value written to X in the largest slot $\hat{s} < s$. If there is no such slot $\hat{s}$, then the value returned is the value of X at the start of the program.

Because all interleavings of processors' instruction streams are possible, our model gains no extra power from allowing a stronger write-conflict resolution protocol such as Priority. In a Priority PRAM, if two or more processors write to the same variable in the same slot, the processor with the highest priority succeeds. However, these writes could also have occurred in successive slots, with the highest priority processor's write occurring first. In that case, a lower priority processor overwrites the value written by the higher priority processor. For the same reasons, our restriction that all reads in a slot occur before all writes in the slot is purely for expositional simplicity. It is possible that all the memory accesses in a given slot could have occurred serially in several slots, and any arbitrary interleaving of these accesses is then possible. Hence, this restriction does not strengthen the model in any way. We use it only to make the discussion simpler.

Since we allow an adversary to select the delays between instruction executions, our model allows all possible interleavings of the processors' instruction streams. Thus, our model is equivalent to an apparently more asynchronous model where atomic reads

and writes occur at arbitrary real points in time. In this respect, our model is similar to those of Kruskal, Rudolph, and Snir [KRS90]; Cole and Zajicek [CZ89], [CZ90a]; and Nishimura [Nis90]. In these models, *all* concurrent accesses (reads or writes) are executed as implied by the usual interleaving semantics of concurrent processes: the outcome of the concurrent execution of several memory accesses is as if these accesses had occurred in some serial order. However, in [KRS90] and [CZ89], [CZ90a], all accesses are executed in one time step. Any program that is correct for our model will run correctly under the assumptions of their models.

We now define a number of important terms.

Termination. In a completely general asynchronous setting, it is not always easy for a processor to know *when* the computation terminates. With each computation, we associate a flag in global memory called *done*, which is initialized to *false*. The asynchronous program terminates when *done* becomes *true*.

Speed functions. We assume the most general form of asynchronous processor behavior. Individual processors can proceed at arbitrarily varying rates of speed during the course of the computation. More formally, we say that each processor, j, possesses its own speed function, which is a possibly infinite ordered list $t_1^j, t_2^j, \ldots$ where t_i^j is the slot in which the jth processor *executes* its ith instruction. (See Fig. 1.)

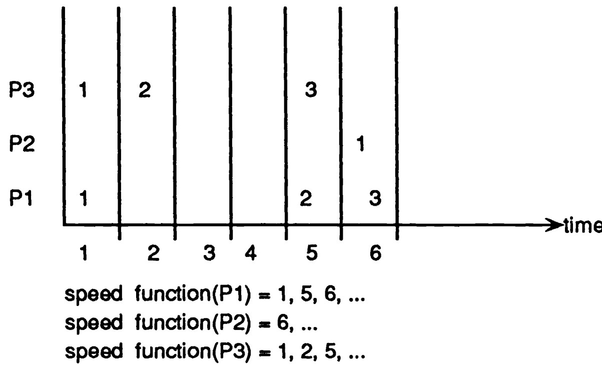


FIG. 1. *Example of processor speed functions.*

Correctness. We say that a program is correct if

- For all possible speed functions, if the program terminates at time t (i.e., *done* is set to *true* at time t), all output locations contain the correct values, and none of these values are changed after *done* is set to *true*. Note that this implies that the program should be correct for all possible interleavings of the processors' instruction streams;
- As long as at least one processor has the capability of doing an infinite amount of work, the program terminates with probability 1.

Work. The *work* of a given run of a program is the total number of single processor instructions performed by the collection of asynchronous parallel processors during that run. This includes any instruction used in busy waiting. Thus, if a program begins in slot T_1 and terminates after slot T_2, each processor working on the program is charged for all instructions it executes during these slots. For each processor j, we charge one for each t_i^j on its list such that $T_1 \le t_i^j \le T_2$. Thus, the work, W, for n processors is $W = \sum_{i=1}^{i=\infty} \sum_{j=1}^{j=n} f(t_i^j)$, where $f(x) = 1$ if $T_1 \le x \le T_2$, and 0 otherwise.

Work is a natural generalization of processor-time product for an asynchronous set-

ting, and is a good measure of an algorithm's efficiency. Kanellakis and Shvartsman [KS89] use a very similar metric (available processor steps) to measure the performance of algorithms on PRAMs with fail-stop errors.

Expected work. We use randomized algorithms to achieve good expected performance for arbitrary asynchronous behavior. For a randomized program, A, we evaluate A as follows. An adversary chooses the most unfavorable set of speed functions for program A. In calculating the expected work for executing A, we assume this worst case choice of speed functions and average over the random choices made by the processors.

Worst case speed functions. We assume processors have no knowledge of their own speed functions and that our results apply for any set of speed functions. However, we require that the processors' speed functions should not be correlated with the results of queries to the random number generators. Thus, our adversary is *oblivious*. The adversary must determine the speed function of each processor before a computation begins. Once a computation commences, the resulting processor speeds are not affected by subsequent queries to the random number generator outputs. If an adversary is allowed to correlate random number generator outputs with speed functions, then it could slow down all processors that choose good work. We discuss the implications of a stronger adversary in §5.1.

Failures. A *fail-stop error* [SS83], [KS89], [KPS90] causes a processor to stop executing, but no incorrect instructions are executed. A processor can fail before or after but not during a write instruction. As long as concurrent writes are Common (i.e., they write the same value), this can be extended to allow failures between bit-writes, using techniques analogous to those used in [KS89]. We allow fail-stop errors in our model. They are modeled formally by a speed function $t_1, t_2, \ldots t_k$, with the fail-stop error occurring at time $t_k + 1$.

Anonymous processors. While it is not essential to our model, all of our algorithms are symmetric, which means that all processors execute the same program and are assumed to be in identical initial states. Because of the random assignment of subtasks to processors, it is not necessary for processors to possess the unique identifiers that are required for a deterministic allocation of processors to subtasks. Parallelization is a consequence of the fact that concurrently executing processors choosing randomly from a pool of work are likely to choose different subtasks.

Notation. We shall use $EO(f(n))$ to denote *expected* $O(f(n))$.

3. An efficient Max finding algorithm. Our interest in algorithms that compute an associative function of n variables arises from the fact that they can be easily adapted to be an efficient synchronization primitive. Since all processor speed functions are possible and A-PRAM processors are anonymous, we need an efficient mechanism by which we can perform n actions and provide a guarantee that they have in fact been performed.

We present Max1, a probabilistic asynchronous algorithm that computes the maximum of n numbers using up to n processors. It uses a binary tree, the leaf nodes of which contain the n numbers. The interior nodes of the tree are initially set to a special reserved value, $\perp$, which indicates that the node is *unevaluated* (see Fig. 2). Algorithm Max1 proceeds as follows.

Processors first examine the root node. If the root node has been evaluated, the algorithm is completed and the processors terminate their program. If the root has not been evaluated, processors simply choose one of the interior nodes at random. If the two children of the chosen node have been evaluated, the node is computed to be the maximum of its two children. The process is then repeated. The root being set indicates that the computation is completed (see Fig. 3).

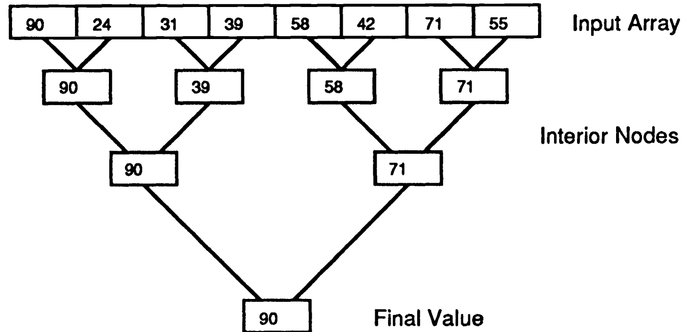


FIG. 2. *Data structure for* Max1.

Each processor executes the following program:

while the root is unevaluated **do**
select an interior node uniformly at random
if the children of the interior node have been evaluated **then** evaluate the node
endwhile

FIG. 3. *Algorithm* Max1.

This randomized task assignment possesses several desirable attributes.

1. Processors are not required to synchronize their operations. It is possible for multiple processors to evaluate the same node simultaneously, but this simultaneous evaluation will not affect the result of the computation. (Of course, redundant evaluation of nodes leads to wasted work, but we will show that the amount of wasted work is relatively small.)

2. It is fault-tolerant. If processors have fail-stop errors, the surviving processors complete the computation. As long as one processor continues to function, the algorithm will make progress toward a solution.

3. Since all processors execute *exactly* the same program, processors do not need to know the identity or the number of processors participating in the computation. Therefore, processors do not need to possess unique identifiers. Processor utilization is improved because processors can join the computation as they become available or leave the computation when they are neeeded elsewhere.

We prove in Theorem 3.1 that Algorithm Max1 computes the maximum of n numbers with $EO(n \log n)$ work. In order to prove Theorem 3.1, we introduce a lemma for analyzing the random evaluation of a graph. The proof of the lemma is in the Appendix.

3.1. A lemma on random evaluation of graphs. Let G be a directed acyclic graph with n nodes. A node with in-degree zero is an *input*, and a node with out-degree zero is an *output*. Let P be the total number of paths that start at some input and end at some output. Let d, the depth of the graph, denote the longest path from an input to an output node.

Initially, only the input nodes are colored and all other nodes are uncolored. Our goal is to color all the nodes of the graph using the following random process. In each

step a set of nodes is selected at random. A selected node is colored in a step if it was uncolored prior to the step, and all of its predecessors have already been colored in prior steps. We require that in each step, each node has at least a constant probability q of being selected. The node selections within a step do not have to be independent. However, the selections in each step must be independent of all other steps. For example, the nodes in a step could be selected by having every node flip a coin with probability q of being selected. Alternatively, we could choose n random integers with replacement in the range $[1 \ldots n]$ and all nodes whose number is chosen are selected, or a single random integer k in the range $[1 \ldots n]$ could be chosen, and the $\frac{n}{2}$ nodes $k, k+1, \ldots k+\frac{n}{2}-1$ (wrapping around to 1 if node ID's greater than n are included) are selected. The following lemma, first proved by Luby [Lub88] in a slightly different form, describes the expected number of steps to color all the nodes.

LEMMA 3.1 (random circuit lemma). *Consider a directed acyclic graph, G, of depth d with P paths from the inputs to the outputs. If in each step, any node that has all its predecessors colored has at least a constant probability $q > 0$ of being colored in this step, then the expected number of steps to color all the output nodes is at most $(6/q)(d + \log P)$, and the probability that more than $(5c/q)(d + \log P)$ steps are used is less than $1/P^c$.*

This lemma essentially states that the expected number of steps to color all nodes is proportional to the depth of the graph as long as the depth is at least as large as $\log P$.

3.2. Creating blocks of work. We now describe a methodology for partitioning an asynchronous computation into blocks that make a minimum amount of progress. Since p processors operate in parallel, it is possible for up to p units of work to be completed simultaneously. It is, therefore, not always possible to divide work into blocks of a fixed size by a delineation in time. We can, however, divide work into blocks that will vary in size by at most p units of work. For example, we can divide work into blocks of size W such that $b \leq W \leq b + p - 1$ units for a fixed minimum block size of b units.

LEMMA 3.2. *Let A be an algorithm that consists of a main loop in which at most k instructions are executed in a complete loop iteration, and let p be the number of processors executing A. Then for any $W \geq pk$, the parallel execution of A can be broken up into consecutive blocks of $\Theta(W)$ work such that each block contains at least W/k complete iterations of the main loop.*

Proof. The first block starts at time zero, and ends at the first time t_1 when the cumulative work is at least $3W$. The next block starts at t_1 and ends at the first time t_2 such that the work in the interval (t_1, t_2) is at least $3W$. The remaining blocks are defined analogously. Since up to p units of work can be performed simultaneously, we cannot divide time into intervals with exactly $3W$ units of work, but we can form intervals that have at least $3W$ units of work and at most $3W + p - 1$ units of work. For a given block, each processor might execute the first instruction of the main loop just prior to the start of the block and then do the rest of the loop within the block. In addition, each processor might execute the first $(k-1)$ instructions of the loop in the block, and the last instruction after the block. Thus a total of at most $2p(k-1)$ units of work can count toward loop iterations, which are not started and completed in the block. Thus a block with at least $3W$ units of work must complete at least $3W - 2p(k-1) \geq 3W - 2W \geq W$ units of work, which counts towards loops started and completed in the block. Dividing by the k instructions per loop iteration yields the theorem. □

Remark. The only part of our proofs where the number of processors and their speed functions plays a role is to guarantee that at least p loops were started and completed in a block of $\Theta(pk)$ work. Processors that do not complete their loop within a block, because of failures or slowdown, can contribute at most $2k(p-1)$ to the wasted work in a block.

Thus, we can allow processors to join the computation or to leave in the middle. The bounds we derive apply as long as the total number of processors that ever participate is $O(p)$. This assumes that instructions executed by a processor when it is not assigned to the computation do not count towards the work used by that computation.

3.3. Analysis of Max1.

THEOREM 3.1. *The* Max1 *algorithm uses* $EO(n \log n)$ *work to compute the maximum of* n *numbers using* $p \leq n$ *processors.*

Proof. By Lemma 3.2, we can define blocks of $\Theta(n)$ work that have at least n loop completions. Each loop completion randomly selects an interior node of the tree. This means each block contains at least n random selections of interior nodes. Since the tree has $n - 1$ interior nodes, and at least n independent random selections in a block, the probability that a given interior node is selected in a block $\geq (e-1)/e$ (Lemma A.7).

Thus, we can consider a block to be analogous to a step of Lemma 3.1, where each node has a positive probability of being selected and evaluated during that block. The number of paths from the leaves of the binary tree to the root is n, and the depth of the tree is $\log n$. Hence, the number of steps to evaluate the tree is $EO(\log n)$. Since each step is equivalent to a block of $O(n)$ work, Max1 requires $EO(n \log n)$ work. □

The Max1 algorithm can be directly extended to compute any associative function of n inputs using up to n processors with $EO(n \log n)$ work.

THEOREM 3.2. *Let* μ *be a constant such that the expected work to complete* Max1 *is* $\leq \mu n \log n$*. Then,* $P[\textit{ work required } \geq k2\mu n \log n] \leq \frac{1}{2^k}$

Proof. Since the work to complete Max1 is a positive random variable, we can apply Markov's inequality. Therefore, each block of $2\mu n \log n$ work has *at least* a $\frac{1}{2}$ probability of completing Max2, *independent* of all other blocks. The theorem follows. □

4. An optimal Max finding algorithm. The maximum of n numbers can be computed sequentially using $O(n)$ work. While Max1 is efficient, it is not optimal since it requires $EO(n \log n)$ work. There exists an optimal PRAM algorithm that computes the maximum of n numbers with $O(n)$ work using $p \leq n/\log n$ processors. This motivated us to find an optimal A-PRAM algorithm.

By using $n/\log n \log^* n$ processors we are able to compute the maximum with $EO(n)$ work. To accomplish this we use a compound data structure that consists of an array of n data elements connected to a binary tree that has $n/\log n$ leaf nodes. The n element data array is divided into $n/\log n$ subarrays of length $\log n$. Each $\log n$ subarray is associated with one of the $n/\log n$ leaves of the tree (see Fig. 4). During the computation, the maximum value of each $\log n$ subarray is computed and placed in its corresponding leaf node. The complete program is specified in Fig. 5.

Note that if a processor selects a leaf node that has been completely evaluated, the processor selects a new node. However, if a processor selects a leaf node that is currently being evaluated by one or more processors, it will have no way of knowing this and will thus also evaluate the leaf node.

4.1. Analysis of algorithm Max2. Algorithm Max2 evaluates a complete binary tree with $n/\log n$ leaves, where evaluating a leaf requires $O(\log n)$ work and evaluating an interior node requires $O(1)$ work. We now show that algorithm Max2 requires $EO(n)$ work using up to $n/\log n \log^* n$ processors. To simplify the analysis, we break up the algorithm into four phases. Initially, the number of unevaluated leaves is $n/\log n$. The first phase ends when the number of unevaluated leaves is less than $n/8 \log n \log^* n$. The second phase ends when the number of unevaluated leaf cells is less than $2n/\log^2 n$. The

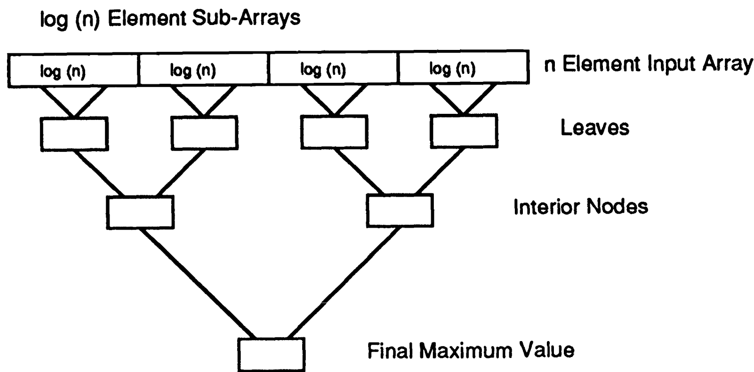


FIG. 4. *Data structure for algorithm* Max2.

Each processor does the following:

while the root node is unevaluated **do**
    choose a tree node at random
    if the node is an unevaluated leaf **then**
        compute the maximum of its $\log n$ element sub-array
    else
        if the node is unevaluated and its two children are evaluated **then**
            evaluate the node
        endif
    endif
endwhile

FIG. 5. *Algorithm* Max2.

third phase ends when all the leaves have been evaluated. The fourth phase starts when all the leaves have been evaluated and ends when the root has been evaluated.

We employ the following strategy throughout the analysis. We break up the computation into blocks. In each block, some amount of progress is made. If the amount of progress exceeds a desired threshold, then we count that block as an *effective* block, else we do not count any progress that occurred in that block for the analysis. We count only the *effective* blocks that are required for the algorithm to terminate. We are able to prove that there is a constant probability of an effective block occurring. So, we know that the expected number of blocks that must occur before we get an effective one is a constant. Thus, counting the expected number of *effective* blocks gives an asymptotic bound on the total number of expected blocks.

In Phases 1 and 2, we break up the work done by all the processors into *blocks* of $\Theta(n/\log^* n)$ work. We then show that the number of blocks needed to reach Phase 3 is $EO(\log^* n)$. It follows that the work to get to Phase 3 is $EO(n)$.

Recall that in a single iteration of algorithm Max2, $O(\log n)$ work is done if a leaf node is evaluated, and otherwise $O(1)$ work is done. As in the proof of Theorem 3.1, we divide time into consecutive intervals such that each interval contains $\Theta(n/\log^* n)$ work. Such an interval is called a *block*. We also divide each processor's instruction stream into *windows*. A window consists of up to $\log n$ node selections and the evaluation of a leaf

if an unevaluated leaf is found. Thus a window consists of $\Theta(\log n)$ work. (A window ends, and the next window starts either after $\log n$ consecutive selections that do not find an unevaluated leaf or after evaluating a leaf.) We now show that each block contains $\Theta(n/\log n \log^* n)$ windows, which both start and complete within the block.

Choose a constant g such that a single processor in one iteration of Max2 requires at most g units of work if no leaf node is found to be unevaluated, and at most $g \log n$ units of work if a leaf node must be evaluated from its $\log n$ element subarray. Thus each window consists of at most $2g \log n$ units of work.

Within a block of $(6gn/\log^* n)$ work, it is possible for each of the $n/\log n \log^* n$ processors to have completed all but the *first* unit of work in a window. It is also possible for each of the $n/\log n \log^* n$ processors to have completed all but the *last* unit of work in a window within this block, meaning that no more than $2 \times (n/\log n \log^* n)(2g \log n - 1) < 4gn/\log^* n$ units of work can be performed that do not count toward the number of completed windows. Thus, at least

$$\frac{6gn}{\log^* n} - \frac{4gn}{\log^* n} = \frac{2gn}{\log^* n}$$

units of work performed by the $n/\log n \log^* n$ processors during a block must count toward windows started and completed within the block.

This means that at least $n/\log n \log^* n$ windows of $2g \log n$ work are started and completed during a block. (Some windows may use less than $2g \log n$ work, but this only increases the total number of windows completed.)

DEFINITION. With respect to a block, we say that a leaf is *new* if it was unevaluated at the start of the block. We say that a window is *successful* if it finds a new leaf. If the new leaf is unevaluated, the remainder of the window will evaluate it. However, a new leaf might have been evaluated by an earlier window in the same block.

Preliminaries for Phases 1 *and* 2. We now show that there is at least a constant probability ($\geq \frac{1}{4}$) that at least $\frac{1}{4}$ of the windows of work in a block are successful.

LEMMA 4.1. *If at least* $2n/\log^2 n$ *leaves are unevaluated at the start of a block, then a block of* $O(n/\log^* n)$ *work has at least a* $\frac{1}{4}$ *probability of having* $n/4 \log n \log^* n$ *successful windows.*

Proof. Since Phase 2 ends when the number of new leaves falls below $2n/\log^2 n$, we can assume that the number of new leaves is greater than or equal to $2n/\log^2 n$. Thus, the new leaves are at least a $(2n/\log^2 n)/(2n/\log n) = 1/\log n$ fraction of the total number of nodes in the tree.

A window can use up to $\log n$ selections to find a new leaf. The probability that a single selection fails to find a new leaf $\leq 1 - (1/\log n)$ (since the new leaves are at least a $1/\log n$ fraction of the total nodes). Define the random variable X_i, such that $X_i = 1$ if the ith window is successful, and 0 otherwise. The probability that a window is successful $= P[X_i = 1] \geq 1 - (1 - (1/\log n))^{\log n} \geq (e-1)/e > \frac{1}{2}$. Thus, during Phases 1 and 2, each window has at least a 50 percent chance of finding a new leaf.

Since a block has $\geq n/\log n \log^* n$ windows, define $X = \sum_{i=1}^{i=n/\log n \log^* n} X_i$ as a lower bound on the number of successful windows in a block. Appealing to well-known probability results in the Appendix (Lemmas A.4 and A.2), $E[X] \geq n/2 \log n \log^* n$ and $P[X \geq n/4 \log n \log^* n] \geq \frac{1}{2}/2 \geq \frac{1}{4}$.

Therefore, there is a $\frac{1}{4}$ probability that a block of $\Theta(n/\log^* n)$ work will have at least $n/4 \log n \log^* n$ successful windows, i.e., they will select and evaluate a new leaf. □

DEFINITION. A block in Phase 1 or Phase 2 is said to be *good* if it contains at least $n/4 \log n \log^* n$ successful windows.

We now calculate the fraction of these successful windows that are nonredundant.

4.1.1. Analysis of Phase 1. If $2s$ random selections are made with replacement from a collection of s items, then it is well known that P[at least $\frac{s}{2}$ distinct items are selected] $\geq 1 - 1/2^s \geq \frac{1}{2}$ (Lemma A.8). Clearly, if the number of items selected from is greater than s, the expected number of redundant selections decreases. By definition each block in Phase 1, starts with at least $n/8 \log n \log^* n$ new leaves. Therefore, it follows that if the block was good, then with probability $\geq \frac{1}{2}$, at least $n/16 \log n \log^* n$ nonredundant selections were made. By Lemma 4.1, there is a $\frac{1}{4}$ probability that a block is good. Thus there is at least a $\frac{1}{8}$ probability that a block will evaluate $\geq n/16 \log n \log^* n$ distinct new leaves.

DEFINITION. A block in Phase 1 is said to be *effective* if at least $n/16 \log n \log^* n$ distinct leaves are evaluated.

After $(16 \log^* n - 2)$ effective blocks, the number of unevaluated leaves

$$\leq \frac{n}{\log n} - (16 \log^* n - 2)\left(\frac{n}{16 \log n \log^* n}\right) = \frac{n}{8 \log n \log^* n}.$$

Since each block has at least a $\frac{1}{8}$ probability of being effective independent of other blocks, we can view blocks as independent Bernoulli trials. Therefore, the expected number of blocks before Phase 1 is over is $\leq 128 \log^* n$ and the work for Phase 1 is $EO(n)$. Note that to simplify the proofs, we have not tried to make the constants tight. The actual constants are much smaller.

Note that in our analysis we use only effective blocks (those that have at least $n/16 \log n \log^* n$ nonredundant selections), discarding progress made in other blocks, even though we pay for their work. While this overestimates the work required to complete, it simplifies the analysis.

4.1.2. Analysis of Phase 2. To simplify the algebra, define $b = n/8 \log n \log^* n$. Phase 2 starts with at most b unevaluated leaves and ends when the number of unevaluated leaves is less than $2n/\log^2 n$.

LEMMA 4.2. *After $EO(\log^* b)$ blocks of $\Theta(n/\log^* n)$ work in Phase 2, the number of unevaluated leaves is less than $2n/\log^2 n$, signaling the end of Phase 2.*

Proof. Consider a good block in Phase 2 in which there are $2b = n/4 \log n \log^* n$ successful windows. Choose k such that $C = b/e^k$ is the number of unevaluated leaves at the start of this block. Define C 0/1 random variables $X_1, X_2, \ldots X_C$, such that $X_i = 1$ if the ith new leaf is not evaluated in this block and 0 otherwise. Since there are $\geq 2b$ successful windows in a good block,

$$P[X_i = 1] \leq \left(1 - \frac{1}{C}\right)^{2b} = \left(1 - \frac{1}{\frac{b}{e^k}}\right)^{2b} = \left(1 - \frac{1}{\frac{b}{e^k}}\right)^{\frac{2b}{e^k} e^k} \leq \left(\frac{1}{e}\right)^{2e^k}.$$

Let $\hat{C} = \sum_{i=1}^{i=C} X_i =$ the number of unevaluated leaves at the end of the block. Thus,

$$E[\hat{C}] = C \times P[X_i = 1] \leq \frac{b}{e^k}\left(\frac{1}{e}\right)^{2e^k} \quad \text{and} \quad P[\hat{C} \leq 2E[\hat{C}]] \leq P\left[\hat{C} \leq 2\frac{b}{e^k}\left(\frac{1}{e}\right)^{2e^k}\right] \geq \frac{1}{2},$$

by Markov's inequality, since $\hat{C}$ is a nonnegative random variable. But, $2(b/e^k)(\frac{1}{e})^{2e^k} \leq (b/e^k)(\frac{1}{e})^{e^k}$ since $e^k \geq 1$. Therefore, $P[\hat{C} \leq (b/e^k)(1/e)^{e^k}] \geq \frac{1}{2}$.

DEFINITION. A block in Phase 2 is *effective* if it is good and if $\hat{C} \le (b/e^k)(\frac{1}{e})^{e^k}$, where $\hat{C}$ is the number of unevaluated leaves at the end of the block and $b/e^k = C$ is the number of unevaluated leaves at the start of the block. We have shown above that a good block is effective with probability greater than or equal to $\frac{1}{2}$.

By Lemma 4.1, the probability that a block in Phase 2 is good is at least $\frac{1}{4}$. Given that a block is good, we have just shown that the probability that it is effective is at least $\frac{1}{2}$. Therefore, the probability that a block in Phase 2 is effective is at least $\frac{1}{8}$.

We now show that after $O(\log^* n)$ effective blocks in Phase 2, the number of new leaves will be less than $2n/\log^2 n$.

For simplicity of analysis, we assume that an effective block makes the minimum amount of progress necessary for it to be deemed effective, i.e., if the number of unevaluated leaves at the start of the jth effective block is $C = b/e^k$, the number of unevaluated leaves at the end of the block is exactly $\hat{C} = (b/e^k)(1/e)^{e^k}$. This assumption can only increase the number of effective blocks necessary to complete Phase 2.

We can now formulate the following recurrence relation

$$\hat{C}_{i+1} = \hat{C}_i \left(\frac{1}{e}\right)^{e^{k_i}}, \quad \text{where} \quad \hat{C}_i = \frac{b}{e^{k_i}}, \tag{1}$$

where $\hat{C}_i$ = number of unevaluated leaves after the ith effective block.

Intuitively, roughly the same number of windows are being concentrated on a rapidly diminishing number of unevaluated leaves. Substituting $\hat{C}_0 = b (\Rightarrow e^{k_0} = 1)$ (this can only increase the number of blocks) in (1) gives $\hat{C}_1 = b(1/e)^1 = b/e$. In calculating $\hat{C}_2$, note that $e^{k_1} = b/\hat{C}_1 = e$.

$$\hat{C}_2 = \hat{C}_1 \left(\frac{1}{e}\right)^{e^{k_1}} = \frac{b}{e}\left(\frac{1}{e}\right)^{e} = \frac{b}{ee^e}.$$

Similarly, $\hat{C}_3 \le b/ee^e e^{e^e} \ldots$. Therefore, within $O(\log^* b) = O(\log^* n)$ effective blocks, the number of unevaluated leaves is less than or equal to $2n/\log^2 n$, at which time Phase 2 ends.

Since each block has at least a $\frac{1}{8}$ probability of being effective, independent of other blocks, we can view blocks as independent Bernoulli trials. Therefore, the expected number of blocks before Phase 2 is over is $EO(\log^* n)$. Since each block contains $\Theta(n/\log^* n)$ work, the total work required for Phase 2 is $EO(n)$.

4.1.3. Analysis of Phase 3. Phase 3 starts with $\le 2n/\log^2 n$ unevaluated leaves and ends when all the leaves are evaluated.

LEMMA 4.3. *Each block of $O(n)$ work in Phase 3 has at least a $1 - (1/n)$ probability of selecting all the unevaluated leaves and the work to complete Phase 3 is $EO(n)$.*

Proof. Call the leaves unevaluated at the start of Phase 3 u-leaves. For simplicity, assume that there are in fact $2n/\log^2 n$ u-leaves. This assumption can only overestimate the work necessary to complete the algorithm. Consider a block of $O(n)$ work in which at least $16n/\log n$ windows of $2g\log n$ work are completed. The probability that a given window fails to find at least one of the $2n/\log^2 n$ u-leaves is $(1-(1/\log n))^{\log n} \le \frac{1}{e} < \frac{1}{2}$.

Following the same reasoning as in Lemma 4.1, we can show that the expected number of windows which find a u-leaf is $\ge 8n/\log n$ and that there is a $\frac{1}{4}$ probability that at least $\frac{1}{4}$ of the windows in a block will find a u-leaf. Therefore, there is a $\frac{1}{4}$ probability that a block has at least $4n/\log n$ windows which select one of the $2n/\log^2 n$ u-leaves.

Since $4n/\log n > 2(2n/\log^2 n)ln(2n/\log^2 n)$, by the well-known coupon collector result (see Lemma A.3 of Appendix), the probability that all the $2n/\log^2 n$ u-leaves are evaluated, given that $4n/\log n$ windows select a u-leaf, is $\geq 1 - \frac{1}{n}$.

Since each block of $O(n)$ work has at least a $1 - (1/n)$ probability of completing all remaining leaves independent of other blocks, the expected number of blocks to complete Phase 3 is a constant. Therefore, Phase 3 requires $EO(n)$ work. □

LEMMA 4.4. *The work required to select and evaluate each of the $n/\log n$ leaves of the binary completion tree in algorithm* Max2 *requires $EO(n)$ work with $\leq n/\log n \log^* n$ processors.*

Proof. This follows as a direct consequence of Lemmas 4.1, 4.2, and 4.3. □

4.1.4. Analysis of Phase 4. In order to prove that the work required to evaluate the completion tree of Max2 is $EO(n)$ once the leaves are evaluated, we shall use Lemma 3.1.

LEMMA 4.5. *The work to evaluate the completion tree of* Max2 *once the leaves are evaluated is $EO(n)$ using $p \leq n/\log n$ processors.*

Proof. By Lemma 3.2, we can define blocks of $\Theta(n/\log n)$ work in which $2n/\log n$ nodes are selected. By Lemma A.7, each node is selected in a block with probability greater than or equal to $(e-1)/e$, since there are less than $2n/\log n$ nodes and $2n/\log n$ selections. Therefore, we can view a block of $\Theta(n/\log n)$ work as equivalent to a step of Lemma 3.1.

The binary tree of Max2 has depth $< \log n$, $\leq 2n/\log n$ nodes and $n/\log n$ paths from the inputs to the output. By Lemma 3.1, the number of steps before the root is evaluated is $EO(\log n)$. Thus, the work required to evaluate the completion tree after the leaves are evaluated is $O(n/\log n) \times EO(\log n) = EO(n)$. □

We are now in a position to state our main theorem.

THEOREM 4.1. *The work to complete* Max2 *is $EO(n)$, using $p \leq n/\log n \log^* n$ processors.*

Proof. This follows as a direct consequence of Lemmas 4.4 and 4.5. □

THEOREM 4.2. *Let μ be a constant such that the expected work to complete* Max2 *is $\leq \mu n$. Then, $P[\text{work required} \geq k2\mu n] \leq 1/2^k$.*

Proof. Since the work to complete Max2 is a positive random variable, we can apply Markov's inequality. Therefore, each block of $2\mu n$ work has *at least* a $\frac{1}{2}$ probability of completing Max2, *independent* of all other blocks. The proof follows. □

COROLLARY 4.1. *Let T be a complete binary tree with L leaves such that processing a leaf requires $W \geq \log L$ work and processing an internal node requires $O(1)$ work and requires that the children have been processed. On an* A-PRAM, *all nodes in T can be processed with $EO(LW)$ work using up to $L/\log^* L$ processors.*

Proof. The intuition is that the work done in finding an unevaluated leaf is subsumed by the work to process it. The proof follows directly from the previous analysis with only the following differences.

Since we have $\leq L/\log^* L$ processors, by Lemma 3.2, we can define blocks in Phase 1 and 2 to consist of $\Theta(LW/\log^* L)$ work. Note that this is asymptotically at least as great as the blocks of $6gL/\log^* L$ work that we defined earlier.

Phase 1 ends when the number of unevaluated leaves is $\leq L/8\log^* L$. Phase 2 ends when the number of unevaluated leaves is $\leq 2L/\log L$. In Phase 3, a block consists of $\Theta(LW)$ work. Phase 3 ends when all leaves have been evaluated. In Phase 4, a block consists of $\Theta(L)$ work.

In Phases 1, 2, and 3, a window consists of W selections to find an unevaluated leaf node, and if one is found, W work to evaluate the leaf node. Since $W > \log L$,

the probability of finding an unevaluated leaf must be at least as great as in the earlier analysis. Hence, the expected number of blocks for each of these phases must be no more than before. Hence, Phases 1 and 2 require $EO(\log^* L)$ blocks, and Phase 3 requires $EO(1)$ blocks.

By substituting $n/\log n$ for L it can be verified that Theorem 4.1 is just a special case of Corollary 4.1. □

4.2. Optimal asynchronous parallel prefix algorithm. Given n numbers $a_1 \circ a_2 \circ \ldots a_n$, and an associative function $\circ$, the parallel prefix problem is to compute: $a_1 \circ a_2 \circ \ldots \circ a_i$ for $i = 1, 2, \ldots n$. The input is an array A of size n. The output is an array *prefix* of size n such that $prefix[i] = a_1 \circ a_2 \ldots \circ a_i$.

Parallel prefix algorithm. We divide the algorithm into two phases. In Phase 1, we break up the input array A into $n/\log n$ blocks of size $\log n$. We construct a complete binary tree, T_1, with $n/\log n$ leaves, each of which is associated with one of the $n/\log n$ blocks of A (similar to Fig. 4). Initially, all the nodes in T_1 are set to $\perp$. Evaluating a leaf requires computing $a_1 \circ a_2 \ldots \circ a_{\log n}$, where $a_1, a_2 \ldots a_{\log n}$ are elements of the block associated with that leaf. Therefore, evaluating a leaf requires $O(\log n)$ work. Evaluating an interior node consists of computing $lchild \circ rchild$, where $lchild$ and $rchild$ are the children of that node. Therefore, evaluating an interior node requires $O(1)$ work.

Processors first evaluate the tree of Phase 1 using the same approach as in Max2. When a processor sees that T_1 is completed, it starts working on Phase 2.

In Phase 2, we break up the output array *prefix* into $n/\log n$ blocks of size $\log n$. We construct a complete binary tree, T_2, with $n/\log n$ leaves, each of which is associated with one of the $n/\log n$ blocks of *prefix*. Initially, all the nodes of T_2 are set to $\perp$. Evaluating a leaf of T_2 entails (i) sequentially computing the parallel prefix of the first element of the block, (ii) sequentially computing the parallel prefix of the remaining elements of the block, and (iii) setting the leaf to 1. Evaluating an interior node involves setting it to 1 if both children have been set to 1, which requires $O(1)$ work.

Using the partial results stored in the nodes of T_1, the parallel prefix for the first element, i, of any block can be computed sequentially with $O(\log n)$ work. For any $i = 1, 2, \ldots, n$, Prefix[i] can be computed using at most one element from each level of T_1, and the binary representation of i determines which elements of T_1 are used. Once the parallel prefix of the first element of a block is known, the parallel prefix of the other elements of the block can be computed sequentially using $O(\log n)$ work. Therefore, evaluating a leaf requires $O(\log n)$ work. The tree for Phase 2 is evaluated using the same approach as in Max2.

THEOREM 4.3. *The parallel prefix of* n *numbers can be computed using* $EO(n)$ *work with up to* $n/\log n \log^* n$ *processors.*

Proof. The tree of Phase 1 is similar to the tree of Max2, where evaluating a leaf requires $O(\log n)$ work and evaluating an interior node requires $O(1)$ work. Therefore, by Theorem 4.1, the work for Phase 1 is $EO(n)$, using up to $n/\log n \log^* n$ processors. Using the same argument, Phase 2 also requires $EO(n)$ work, using up to $n/\log n \log^* n$ processors. □

5. Simulations. The algorithms of §§3 and 4 are useful because they serve as good building blocks. However, these are specific results, and it is of greater interest to find more general results. A large body of work has been done on the PRAM model, which is a clean and elegant model for parallel algorithm design and analysis. To develop and analyze an asynchronous equivalent of each PRAM algorithm of interest is an unattractive choice.

Instead, we would like to identify those PRAM algorithms that are amenable to automatic translations into their A-PRAM equivalent. In this section we will show that the algorithms we developed in §§3 and 4 can be used to efficiently simulate a restricted class of PRAM computations on an A-PRAM.

On a PRAM, synchronization between steps is achieved implicitly. However, for an asynchronous system we must explicitly detect the termination of a step before allowing processors to proceed to the next step. The problem of completing a PRAM step and detecting that the step has been completed can be abstracted as the *Certified Write All Problem* (CWA), which is defined as follows: given an array $a[1..n]$ and a flag f both initialized to 0, set $a[i] = 1$, for all $i = 1, 2, \ldots, n$, and then set $f = 1$ after all array elements are 1.

Algorithms Max1 and Max2 described in the prior sections can be easily adapted to solve the CWA problem with $EO(n \log n)$ work using up to n processors and with $EO(n)$ work using up to $n/\log n \log^* n$ processors.

Related work. Kanellakis and Shvartsman [KS89], [Shv89] and Kedem, Palem, and Spirakis [KPS90] investigated simulations of PRAM programs on PRAMs, which allowed fail-stop errors, but which work synchronously until an error occurs. We call this type of PRAM a C-PRAM (for *Crash*). In [KS89], deterministic simulations of a limited class of PRAM programs on C-PRAMs were given. In [Shv89], [KPS90], it was shown that a C-PRAM can simulate each step of any PRAM program using the work necessary to solve the CWA problem. Thus, the solution to the CWA problem given in §4, combined with the [KPS90] simulation technique, allows a C-PRAM to simulate any n processor CRCW PRAM program using $EO(n)$ work per step and up to $n/\log n \log^* n$ simulating processors. Thus, our CWA algorithm gives an optimal simulation on a C-PRAM with only a modest reduction in the number of processors that can be used.

A C-PRAM allows only limited asynchronous behavior. However, in [MSP90], we show that the KPS simulation technique can be extended to allow optimal simulations of any CRCW PRAM program on a *loosely atomic* A-PRAM. A loosely atomic A-PRAM has a special *Fetch-Test-Store* instruction (FTS). The FTS instruction is not atomic, but is loosely atomic, which means that if a processor executes the Fetch instruction at time t_1, the Test at time t_2, and the Store at time t_3, the total work by all processors between time t_1 and t_3 must be at most n. Note that if there are $n/\log n \log^* n$ processors, then it is sufficient for loose atomicity that no processor be more than $\log n \log^* n$ faster than the slowest processor.

The simulation in [MSP90], however, allows any n processor CRCW PRAM program to be simulated on a loosely atomic A-PRAM with $EO(n)$ work per step using up to $n/\log n \log^* n$ processors. The restriction of loose atomicity can be relaxed by adding other features to the model such as a broadcast facility or tags [MSP90], [SM90] or by using extra space [BMP91].

The simulations we have described require some limitations on the A-PRAM model. We will now show that a restricted class of computations can be simulated optimally by an A-PRAM with no restrictions. We start by discussing a problem that must be overcome to allow simulations on A-PRAMs. With completely asynchronous processor behavior, it seems essential to allow more than one processor to execute the same task in order to get efficient computations. However, this creates a setting where it is very difficult to reuse memory. If a memory cell's proper value is v_1 at one point of the computation and v_2 at a later point, the following may occur: The cell is set to v_1 by processor P_1 at time t_1; processor P_2 is also ready to write v_1 at t_1, but suffers a long delay. The cell is then set to v_2 by processor P_1 at time t_2; processor P_2 finally commits its write of v_1 at $t_3 > t_2$,

thereby overwriting the correct value. The cell is then read by processor P_3, and it gets the wrong value v_1.

To avoid the problem described above, we define a class of computations called *Computation Circuits* (C-Circuits), which are highly structured. For these computations we will be able to create programs with the property that each memory cell has a unique value written to it in a given program execution. A Computation Circuit is similar to a bounded degree fan-in circuit [KR90], [SV84]. Computation nodes represent simple computations (like gates in a normal circuit), and the data nodes are inputs to the C-Circuit, outputs of the C-Circuit, or intermediate values.

Formally, a C-Circuit is a directed acyclic graph with two types of nodes: *data nodes* and *computation nodes*. A subset of the data nodes are *input nodes* that have indegree zero and *output nodes* that have outdegree zero. All arcs in the graph go from data nodes to computation nodes or from computation nodes to data nodes. Each data node, except for the input nodes, has indegree one and arbitrary outdegree. Each computation node has bounded indegree and outdegree. Each computation node N has associated with it a constant length sequence of RAM instructions. These instructions read from the data nodes that have arcs directed into N. They write a value to each data node that has an edge directed from N. Thus each computation node represents a function computable in constant time, whose inputs are the immediate predecessors of N and whose outputs are the children of N. Initially, all the input nodes contain the input values to the C-Circuit, and all other data nodes contain a special value $\perp$, which indicates that they have not yet been written to.

The *size* S of a C-Circuit is the number of computation nodes, and the *depth* D is the longest path from an input to an output. For a parameter p, the C-Circuit can be divided into $L= D + \lceil S/p \rceil$ layers [Bre74] such that:

1. Each layer contains computation nodes and all their children;
2. Each layer has at most p computation nodes;
3. All computation nodes in layer one have only inputs as predecessors; and
4. All the predecessors of computation nodes in layer $i, i = 2, 3, \ldots, L$ are in lower numbered layers or are inputs.

All the computation nodes in layer one have all their input values, so it is easy for p PRAM processors to compute all the values for the children of these computation nodes in $O(1)$ time. In general, after all the computation nodes in layers $1, 2, \ldots, i-1$ are processed, then all the computation nodes in layer i have their input values, and it is easy for p PRAM processors to compute the values of all the children of computation nodes in layer i in $O(1)$ time. Thus, once the C-Circuit is divided into L layers, it is easy to create a p processor CREW PRAM program that processes computation nodes layer by layer and produces the outputs in $O(L)$ time. The processor-time product is $Lp = Dp + S$. This is a time optimal direct simulation of the C-Circuit with p processors on a CREW PRAM since both D and S/p are lower bounds on number of parallel steps for a direct simulation. If $p \leq S/D$, then the processor-time product is $O(S)$, which is optimal.

We can also create an A-PRAM algorithm that computes the outputs of the C-Circuit, using $EO(Lp)$ work and up to $p/\log p \log^* p$ processors.

THEOREM 5.1. *Given a* C-Circuit *of size S and depth D, then there is an* A-PRAM *program that computes the outputs of the* C-Circuit *using $EO(S)$ work and up to $p/\log p \log^* p$ processors, where $p = \lceil S/D \rceil$.*

Proof. Partition the C-Circuit into $L = D + \lceil S/p \rceil$ layers with properties (1)–(4) above. We associate with each layer $i, i = 1, 2, \ldots, L$ of the C-Circuit a complete binary

tree T_i with $p/\log p$ leaves. Each leaf is associated with a block of $\log p$ computation nodes. Processors choose random nodes in T_i using essentially the same algorithm used in Max2, except that when an unselected leaf node is chosen, the processor processes each of the $\log p$ computation nodes associated with that leaf, then sets the leaf to 1 (if there are fewer than p nodes in a layer some leaves may have fewer than $\log p$ computation nodes associated with them). The analysis in Theorem 4.1 shows that the expected work to process all the computation nodes in layer i, and set all the nodes in T_i to 1 is $EO(p)$ using up to $p/\log p \log^* p$ A-PRAM processors.

The preceding description processes a single layer. For the complete computation, each A-PRAM processor does the following:

repeat
Find the smallest index i such that the root of T_i is not set
Work on tree T_i until the root of T_i is set to 1.
until the root of the last tree, T_L, is set to 1.

Each processor does $O(L)$ total work searching for the correct tree to work on, and does at most $O(\log p)$ work on each tree after its root is set to 1. Thus the total expected work is dominated by the work to process each tree. Therefore, the total work is $EO(Lp) = EO((D + \lceil S/p \rceil)(p)) = EO(Dp + S) = EO(S)$, since $p = \lceil S/D \rceil$. □

Any function computed by a bounded degree fan-in circuit of size S and depth D can be easily computed by a C-Circuit, which has size S and Depth D. Thus Theorem 5.1 applies to any function computable by a bounded degree fan-in circuit. In fact, Stockmeyer and Vishkin [SV84] have shown that any n input function computable by a Priority CRCW PRAM (whose arithmetic instructions are restricted to addition and subtraction) in time T using p processors can be computed by a bounded degree fan-in circuit whose size is polynomial in p, n, and T and whose depth is $O(T)$. The [SV84] result combined with Theorem 5.1 shows that it is possible to convert any CRCW PRAM program to an asynchronous program. However, the polynomial increase in work usually makes this an impractical simulation.

5.1. Work and time. In this section we consider two additional issues related to the analysis of our randomized A-PRAM algorithms. We first discuss the possibility of poor performance by our algorithms, and we then consider the time complexity of our algorithms.

Since we use randomized algorithms there is always the possibility that a particular run of the algorithm will use much more work than the algorithm's expected work. We have shown in Theorems 3.2 and 4.2 that the distribution of the work taken by our algorithms is tightly centered around the mean. However, in their current form, our algorithms can take unbounded work for a pathological set of random choices by the processors. A related issue is the expected performance of our algorithms against an adaptive adversary, which can set the speed functions as it sees the random selections made by the algorithm. More formally, an adaptive adversary sees the random selections made by the processors in time slot i and then determines for each processor the time slot in which it will execute its next instruction. It is fairly easy to construct adaptive adversaries that force our CWA algorithms to take $\Omega(n^2/\text{polylog}(n))$ expected work [KS91]. From a system point of view, an adaptive adversary models a setting where the speed of a processor depends on the instructions it selects. An oblivious adversary represents a setting where the speed function of the processor is independent of the instructions it executes.

It is possible to use a standard technique to achieve the good expected times of the algorithms we have presented, while guaranteeing that the worst case performance will not be too bad. We can have our processors simultaneously execute both our randomized algorithm and a deterministic algorithm (thus each processor executes the first instruction of the randomized algorithm, then the first instruction of the deterministic algorithm, then the second instruction of the randomized algorithm, ...). As soon as either the randomized or the deterministic algorithm completes, the overall algorithm halts. This will at most double the expected time of the randomized algorithm and the worst case time of the deterministic algorithm. For the CWA problem there is a recent deterministic algorithm due to Buss and Ragde [BR90], which uses $O(n^{\log_2 3})$ work using up to n processors. (Essentially the same algorithm was described in [KS91].) Thus we can combine this deterministic CWA algorithm with our randomized CWA algorithm to get an algorithm that uses linear expected work, while using $O(n^{\log_2 3})$ work in the worst case. Recently, Anderson and Woll [AW91] have shown that for any $\epsilon > 0$ there exists a deterministic CWA algorithm that uses $O(n^{1+\epsilon})$ work using up to n processors. Thus it should be possible to get even better worst case bounds for algorithms that combine randomized and deterministic solutions to the CWA problem.

5.2. Time complexity. We now consider the expected time to complete an asynchronous algorithm when we make some assumptions about the algorithms and about the processor speeds. For an asynchronous algorithm $\mathcal{A}$, its expected work will depend on the input size n and the number of processors p used. Let $W(n,p)$ denote the expected work to complete algorithm $\mathcal{A}$ on an input of size n using p processors. We define an algorithm $\mathcal{A}$ to be *persistent* if there exists a constant c such that an interval of time that completes at least $cW(n,p)$ work completes $\mathcal{A}$ with probability at least $\frac{1}{2}$ regardless of the state of the algorithm at the start of the interval. All the algorithms discussed in this paper are persistent.

In order to make statements about the expected time of an algorithm we must have some indication of the time required for a minimum amount of work to be completed. Suppose that we have an A-PRAM with associated constants $d \leq 1$ and an integer $\tau \geq 1$ such that in each time interval of length τ, $[1,\tau]$, $[\tau+1, 2\tau]$, ..., the total work performed by p processors is at least $dp\tau$ with probability $\geq \frac{1}{2}$. We assume this is true in every interval of length τ independent of the work performed in other time intervals. Note, however, that the work can be distributed very unevenly within the interval. We will call such a processor system *fixed rate* with parameter τ.

THEOREM 5.2. *Any persistent algorithm $\mathcal{A}$ that uses expected work $W(n,p)$ runs in expected time $O(W(n,p)/p)$ on a p processor* A-PRAM, *which is fixed rate with parameter τ.*

Proof. Let c be the constant such that each interval of $cW(n,p)$ work completes $\mathcal{A}$ with probability at least $\frac{1}{2}$, and let $\lceil 2cW(n,p)/\tau pd \rceil = \Delta$. Since the expected work completed in an interval of length τ is at least $\tau dp/2$, Δ is the number of intervals of length τ to complete $cW(n,p)$ units of work if each interval contains its expected amount of work. The expected work completed in each interval $[1, 2\Delta\tau]$, $[2\Delta\tau + 1, 4\Delta\tau]$, ..., is at least $2cW(n,p)$. Thus by Lemma A.2, each such interval of length $2\Delta\tau$ completes at least $cW(n,p)$ units of work with probability at least $\frac{d}{4}$. Since $\mathcal{A}$ is persistent, any interval containing at least $cW(n,p)$ work completes the algorithm with probability at least $\frac{1}{2}$. Thus each interval of length $2\Delta\tau$ completes $\mathcal{A}$ with probability at least $\frac{d}{8}$. The expected number of intervals of length $2\Delta\tau$ to complete $\mathcal{A}$ is thus $\frac{8}{d}$, a constant, so the expected running time of $\mathcal{A}$ is $O(\Delta\tau) = O(W(n,p)/p)$. □

There are many reasonable settings in which an A-PRAM would be fixed rate. One

simple setting is where each processor completes at least one instruction every t time units for some constant t. Clearly such a system is fixed rate with parameter t. Two other settings of interest are the unbounded delays and bounded delays models of Cole and Zajicek [CZ90b]. In the unbounded delays model each processor does its next instruction in the current time slot with probability q, and does nothing with probability $1-q$. Thus if we set $\tau = \lceil \frac{2}{q} \rceil$, then the expected work in an interval of length τ is $qp\tau > 2p$. If we set $d = \frac{q}{2}$, then each interval of length τ does at least $dp\tau$ work with probability at least $\frac{1}{2}$.

In the bounded delay model of [CZ90b], each processor executes its next instruction in the current time interval with probability $1-q$, and with probability q executes its next instruction after a delay of $k-1$ time units (so the instruction effectively takes k time units). If the parameters q and k are constants that do not depend on the input size or the number of processors, then a bounded delay A-PRAM is fixed rate with parameter $2k$ and $d = \frac{1}{2k}$.

6. List ranking and sorting. In this section we show that we can apply our previous results to get good asynchronous algorithms for list ranking and sorting. This provides a very strong set of primitives for building efficient asynchronous algorithms. Our algorithms for both list ranking and sorting require $\Theta(n \log n)$ space rather than the $O(n)$ space used by the synchronous parallel algorithms. The increase in space is largely due to the difficulty of reusing space in an asynchronous setting.

6.1. A simple asynchronous list ranking algorithm. The list ranking problem is: given a linked list, we would like to determine, for each element in the list, its distance from the tail. The sequential algorithm takes $O(n)$ time. We make the standard assumption that the list is stored in an array of n contiguous locations, which contain the records that comprise the linked list in some arbitrary order. This facilitates assignment of processors. We now describe an efficient asynchronous algorithm for list ranking.

Input. An array *link* that contains the original links. An array *rank* initialized as $\forall i, i = 1, 2, \ldots, n,\ rank[i] = \perp$, except for *end_of_list* whose rank is 0.

Output. At the end of the computation, each element of the list knows its distance from the end of the list, entered in the rank array.

Wyllie proposed a simple algorithm (see Fig. 6) that solves the list ranking problem on an EREW PRAM in $O(\log n)$ time with n processors [Wyl81] . It uses an array *length* that contains the lengths of the links, which are initially 1. The operation of replacing each pointer, *link*[*i*], by the pointer's pointer, *link*[*link*[*i*]], is called *pointer jumping*. Note that this is not an optimal algorithm since the work is $O(n \log n)$.

More sophisticated techniques have been proposed [MR85], [CV89], [AM89], which solve the problem in $O(\log n)$ time and $O(n)$ work using $n/\log n$ processors.

The precedence constraints of the computation can be viewed as a directed acyclic graph (DAG), G, as shown in Fig. 7. The top layer of the graph is the links of length 1, the second the links of length 2, then 4,..., up to $n/2$. The bottom layer is the output of the computation, the ranks. The left column represents the links computed for the head of the list. The next column represents the links computed for its successor, and so on. The right column represents the links computed for the tail of the list.

We cannot construct the DAG of Fig. 7 prior to the computation since it depends on the ranks. Instead, we construct a two-dimensional array *Pointer* of size $\log n$ by n which is set to $\perp$, except $\forall j, j = 1, 2, \ldots, n,\ Pointer[0, j] = link[j]$, the location at a distance of 1 from j in the linked list. The ith row of *Pointer* represents the ith layer of the DAG G (but the jth column of *Pointer* need not represent any specific positions of

```
for log n iterations do
    in parallel for each processor i ← 1 to n do
        if rank[i] = ⊥ then
            if rank[link[i]] = ⊥ then
                length[i] ← length[i] + length[link[i]]
                { If rank of link[i] not known, double pointer's length }
                link[i] ← link[link[i]] { pointer jump }
            else
                rank[i] ← length[i] + rank[link[i]]
                { If rank of link[i] known, compute rank[i] }
                link[i] ← link[link[i]] { pointer jump }
                { Update pointer for that element }
            endif
        endif
    endfor
endfor
```

FIG. 6. *Wyllie's* PRAM *list ranking algorithm.*

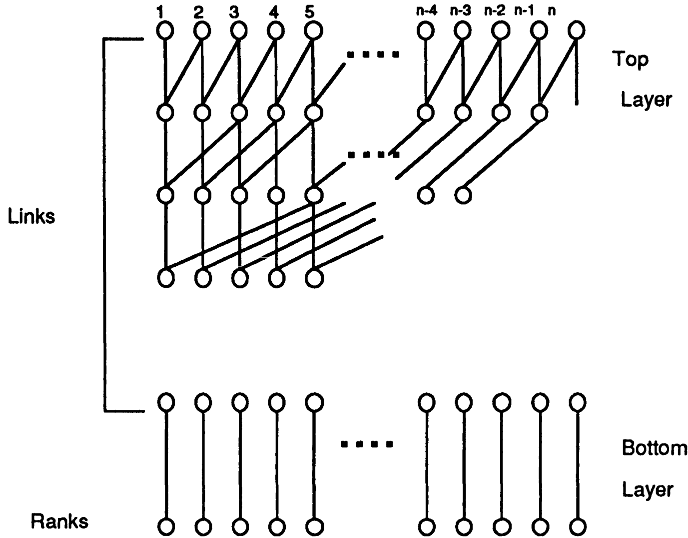


FIG. 7. DAG *corresponding to algorithm* LR0.

the DAG). Therefore, $Pointer[i, j]$ corresponds to some node k in layer i of the DAG G. The key point is that performing the computation associated with node k in layer i of the DAG G does not require knowing that it corresponds to $Pointer[i, j]$. Therefore, we can view $Pointer$ as representing nodes of a $\log n$ depth DAG of width n.

Outline of algorithm. We now show how the entries in $Pointer$ are computed. (The detailed program is in Fig. 8.) During the course of the computation, the entries of $Pointer$ will be filled in such that $Pointer[i, j]$ points to the location at a distance of 2^i from j. For $i = 0, 1, \ldots \log n, j = 1, \ldots n$, $Pointer[i, j]$ holds the link from node j to the

node at a distance of 2^i from it, should such a node exist. If $Pointer[i, j]$ is selected, we try to compute the pointer of length 2^i from j. This can be done only if the pointer of length 2^{i-1} from j (say this points to k) has been computed and the pointer of length 2^{i-1} from k has been computed. In other words, $Pointer[i-1, j]$ and $Pointer[i-1, k]$ must have been computed, where $Pointer[i-1, j] = k$. If $Pointer[i-1, j] = k$ and $rank[k]$ is known, then we can compute the rank of k, and no entries of $Pointer[r, j]$ for $r > i-1$ are ever computed.

```
while the computation has not terminated do
    Select an entry in Pointer[i, j] at random
    if Pointer[i − 1, j] is defined (say, Pointer[i − 1, j] = k) then
        if rank[k] is defined (say, rank[k] = r) then
            rank[j] ← 2^{i−1} + r
            { node j is at distance 2^{i−1} + r from the tail }
        else { rank[k] is undefined }
            if Pointer[i − 1, k] is defined then
                Pointer[i, j] ← Pointer[i − 1, k]
            endif
        endif
    endif
endwhile
```

FIG. 8. *Asynchronous list ranking algorithm* LR0.

THEOREM 6.1. *List ranking can be done in $EO(n \log^2 n)$ work with up to $n \log n$ processors.*

Proof. We have shown that computing the elements of *Pointer* is equivalent to computing the DAG of Fig. 7, which corresponds to Wyllie's algorithm for list ranking. By Lemma 3.2, we can define a block of $\Theta(n \log n)$ work, in which $n \log n$ random entries in *Pointer* are selected and evaluated if possible. By Lemma A.7, each node in the DAG of Fig. 7 has at least a constant probability of being selected in a block. Hence, a block is equivalent to a step of Lemma 3.1. The DAG has depth $\log n$ and $O(n^2)$ paths. By Lemma 3.1, $EO(\log n)$ steps are required to evaluate the DAG. Hence, the work for LR0 is $\Theta(n \log n) \times EO(\log n) = EO(n \log^2 n)$ using $p \leq n \log n$ processors. □

Termination detection. We use a complete binary tree, T, whose leaves are the $\log n$th row of *Pointer* and the Max1 algorithm of Theorem 3.1 to detect when all values have been computed. Initially, all nodes of T are set to $\perp$. The ith leaf of T is set to *done* when $rank[i]$ has been computed. An interior node of T can be set to *done* when both its children have been set to *done*. The computation terminates when the root of T is set to *done*. Processors alternate between selecting list items in *Pointer* and nodes in T. Since T has $O(n)$ nodes, n paths and depth $\log n$, this at most doubles the work.

Note that this algorithm for list ranking, like Wyllie's, can easily be extended to the more general case where the links have lengths other than 1. In this case, when we "pointer jump," the length of the new pointer is the sum of the lengths of the two pointers used in its creation.

List ranking algorithm LR1. We can improve the efficiency of LR0 by reducing the number of processors to $n/ \log n \log^* n$. We now introduce algorithm LR1. It is similar to LR0 except that we evaluate *Pointer* row by row. To ensure that a given row has been completely evaluated before we proceed to the next row, we use a completion tree, similar to Fig. 4 used in Max2. Letting a row of *Pointer* be the input array for

completion tree, evaluating a row of *Pointer* and certifying that it has been completely evaluated requires $EO(n)$ work using $p \leq n/\log n \log^* n$ processors, by Theorem 4.1. Since *Pointer* has $\log n$ rows, LR1 requires $EO(n \log n)$ work using $p \leq n/\log n \log^* n$ processors.

THEOREM 6.2. *List ranking can be done in $EO(n \log n)$ work with up to $n/\log n \log^* n$ processors.*

List ranking algorithm LR2. Along each column of the DAG, there is at most one node, whose selection could lead to useful work. Hence, there are at most n nodes whose selection could lead to useful work. This observation leads us to drop the maximum number of processors to n.

We define an entry $Pointer[i, j]$ as a $frontier$ node if $Pointer[i, j]$ is unevaluated, but $Pointer[i - 1, j]$ is evaluated. In order to confine our selections to the $frontier$ nodes, instead of picking nodes entirely at random, each processor picks columns at random. Having picked a column, say j, we perform binary search along the jth column to find the first entry, say $Pointer[l, j]$, which has not yet been computed. (Note that the entries along a column must be filled consecutively, i.e., if $Pointer[i, j]$ is defined, then $\forall k, k < i, Pointer[k, j]$ must also be defined.) If $Pointer[l, j]$ can be computed, we do so. Thus we get the following.

THEOREM 6.3. *List ranking can be done in $EO(n \log n \log \log n)$ work with up to n processors.*

Proof. Since $p \leq n$, by Lemma 3.2, we can break up the computation into blocks of $\Theta(n \log \log n)$ work, consisting of

1. selecting n, not necessarily distinct, columns in the *Pointer* array;
2. searching the column to find the first uncomputed node; and
3. computing this node if possible.

Since a column of *Pointer* has $\log n$ entries, the binary search along it takes $O(\log \log n)$ work.

With respect to a block, we define a $frontier$ node as one which is a frontier node at the start of the block. By Lemma A.7, each column of *Pointer* has at least a constant probability of being selected in a block. Since there can be at most one frontier node in a column, each frontier node has at least a constant probability of being selected and computed in a block. Since only frontier nodes can make progress, each block of $O(n \log \log n)$ work corresponds to one step of Lemma 3.1.

The circuit of Fig. 7 has $O(n^2)$ paths from the inputs to the outputs and depth $\log n$. Thus, by Lemma 3.1, after $EO(\log n)$ blocks all the outputs will have been computed, and the total work is $EO(n \log n \log \log n)$. Detecting completion is similar to the Certified Write All problem where one writes a 1 into $A[i]$ if $rank[i]$ has been computed. With the $\log n$th row of *Pointer* serving as the input array, Max2 can be used to determine whether all input entries have been computed. By Theorem 3.1, Max1 requires $EO(n \log n)$ work using $p \leq n$ processors. Processors alternate between selecting list items in *Pointer* and nodes in T, which at most doubles the work. □

This last result demonstrates that asynchronous, randomized algorithms need not always select work completely at random. A search for useful work may yield improved efficiency.

The circuit of Fig. 7 is of size $\Theta(n \log n)$, so no direct computation of it can take $o(n \log n)$ work. However, there are alternate PRAM algorithms for list ranking which take $O(n)$ work using up to $n/\log n$ processors [CV89], [AM89], and it would be of interest to try and extend these more sophisticated algorithms to our model. In [MS90] we describe a more efficient A-PRAM list ranking algorithm. It is more complicated,

but uses only $EO(n \log \log n)$ work with up to $n/\log n \log^* n$ processors.

6.2. Sorting. The technique of §6.1 can be applied with some modifications to the $O(\log n)$ depth sorting network of Ajtai, Komlos, and Szemeredi [AKS83]. The computation can be viewed as a circuit with n comparator nodes at each level, each of which has two inputs and two outputs. This circuit can be evaluated in $O(\log n)$ time with n processors by a simple PRAM algorithm.

THEOREM 6.4. *Using the* AKS *sorting network, sorting can be done with* $EO(n \log^2 n)$ *work using up to* $n \log n$ *processors.*

Proof. Since we have $n \log n$ processors, we can define a trial of $\Theta(n \log n)$ work that consists of selecting and evaluating, if possible, $\Theta(n \log n)$ comparator nodes. In a block of $\Theta(n \log n)$ work, each node in the circuit has at least a constant probability of being selected, by Lemma A.7. Hence, we can view a block as equivalent to a step of Lemma 3.1. The AKS sorting network is equivalent to a circuit of depth $O(\log n)$, and number of paths is polynomial in n ($= n \times 2^{c \log n}$). By Lemma 3.1, $EO(\log n)$ steps are required to evaluate the circuit. Since a block of $\Theta(n \log n)$ work is equivalent to a step, it follows that sorting can be done with $EO(n \log^2 n)$ work using $p \leq n \log n$ processors. □

THEOREM 6.5. *Using the* AKS *sorting network, sorting can be done with* $EO(n \log n)$ *work with up to* $n/\log n$ *processors.*

Proof. As in Theorem 6.2, we can evaluate the circuit row by row, by placing a completion tree over each row. The proof follows directly. □

The expected work with n processors can be improved in a manner analogous to LR2 (in Theorem 6.3) as follows.

Data structures. The input is the sorting network, with $2n$ data items at the inputs of the n comparator nodes at the top level. The output is the outputs of the bottom level of the sorting network, which contain the data items in sorted order.

Let $c \log n$ be the depth of the sorting network. We define a two-dimensional array *Network* of size $c \log n$ by n such that $Network[i, j]$ corresponds to comparator node j in level i of the sorting network. Each entry of Network contains two fields, $port_1$ and $port_2$, where the data items to be processed by this node will be placed. Each data item is associated with its index in the input array, which is used in conjunction with the *Progress* array defined below, to trace its progress through the network. We create a two-dimensional array called *Progress* of size $c \log n \times 2n$. Initially, $Progress[0, i] = \lceil i/2 \rceil, i = 1, 2, \ldots n$, all other entries being $\perp$. What this means is that the ith data item is at the input of comparator node $i/2$ of the 0th row. The first entry is the layer of the circuit that it has reached, and the second is the index of the data item, e.g., $Progress[i, j]$ is the comparator node that gets data item j in level i. During the course of the computation, the entries will be filled in such that if $Progress[i, j] = m \neq \perp$, then $Progress[l, j] \neq \perp$ for $l = 0, 1, \ldots, i-1$. Further, it means that the jth data item is an input of the mth comparator in level i of the circuit.

Algorithm. Each A-PRAM processor does the following. While the computation has not terminated, select one of the data items at random. Let this be j. If its final sorted position has not been determined, then perform binary search along the jth column, from $Progress[0, j]$ to $Progress[c \log n, j]$ to find the last filled entry. Let the last filled entry be i, and let $Progress[i, j] = m$. This implies that the jth data item is an input to the mth comparator node in level i of the sorting network.

If $Network[i, m].port_1 = \perp$ or $Network[i, m].port_2 = \perp$, then abandon this random selection since both inputs to the comparator node $Network[i, m]$ are not available.

If $Network[i, m].port_1 \neq \perp$ and $Network[i, m].port_2 \neq \perp$, it means that the other input, say the kth data item, of this comparator node is available. In that case, we per-

form the comparison associated with that comparator node. As a result of this comparison, suppose the jth data item is to be placed in the first port of the pth comparator node of level $i + 1$, and the kth data item is to be placed in the second port of the qth comparator node at the level $i + 1$. Then, we make these entries in *Network* as follows: $Network[i + 1, p].port_1 \leftarrow j$ and $Network[i + 1, q].port_2 \leftarrow k$. We update the *Progress* entries for those two data items as follows: $Progress[i + 1, j] \leftarrow p$ and $Progress[i + 1, k] \leftarrow q$.

We use a complete binary tree, T, whose leaves are the $c \log n$th row of *Network* and the CWA algorithms of §§3 and 4 to detect when all values have been sorted. As in LR2, processors alternate between selecting data items in *Network* and nodes in T, which at most doubles the work.

THEOREM 6.6. *Sorting can be done in $EO(n \log n \log\log n)$ work with n processors.*

Proof. Since $p \leq n$, by Lemma 3.2, we can define blocks of $\Theta(n \log\log n)$ work consisting of

1. selecting n, not necessarily distinct, columns in the *Progress* array;
2. searching the column to find the first uncomputed node; and
3. computing this node if possible.

Since a column of *Progress* has $\log n$ entries, the binary search along it takes $O(\log\log n)$ work. We define a *frontier* node as one which is itself unevaluated and has at least one of its inputs evaluated. There can be at most $2n - 1$ frontier nodes at any time. In a block, each data item, and hence each frontier node, has at least a constant probability of being selected. Thus, as in Theorem 6.3, each block of $O(n \log\log n)$ work corresponds to one step of Lemma 3.1. The sorting circuit has $O(n2^{c \log n})$ paths from the inputs to the outputs, and has a depth of $O(\log n)$. By Lemma 3.1, after $EO(\log n)$ steps, all the outputs have been computed, and the total work is $EO(n \log n \log\log n)$. □

7. Conclusions. We have shown in this paper that several fundamental problems can be solved efficiently on an A-PRAM. In addition, the techniques in this paper provide the tools to convert a large class of PRAM programs to optimal asynchronous programs. We consider these techniques to be an important step in creating a bridge between PRAMs and more realistic parallel machines.

There are still many open questions in this area. One of the most important issues relates to initializing global memory. Our algorithms and the vast majority of other algorithms in this area assume that we start with global memory initialized to some special value. We know of no way in which a system of A-PRAM processors can initialize global memory without some restrictions on the asynchronous behavior of the processors. An interesting and important area of future research would be to identify restrictions on the asynchronous processor behavior which allow global memory to be initialized efficiently.

A second important issue is the development of work-optimal deterministic CWA algorithms. The recent results of Buss and Ragde [BR90] and Anderson and Woll [AW91] suggest possible directions for developing work-optimal deterministic CWA algorithms.

Appendix. We now prove Lemma 3.1 for analyzing the random evaluation of a graph, which was introduced in §3. This lemma and its proof are based on similar results due to Luby [Lub88].

Let G be a directed acyclic graph with n nodes. A node with in-degree zero is an *input* and a node with out-degree zero is an *output*. Let P be the total number of paths that start at some input and end at some output. Let d, the depth of the graph, denote the longest path from an input to an output node.

Initially, only the input nodes are colored and all other nodes are uncolored. Our

goal is to color all the nodes of the graph using the following random process. In each *step* a set of nodes is selected. A selected node is colored in a step if it was uncolored prior to the step, and all of its predecessors have already been colored in prior steps. The selected nodes are chosen at random. We require that in each step, each node have at least a constant probability q of being selected. (Note: the node selections within a step do not have to be independent. However, the selections in each step must be independent of all other steps. For example, the nodes in a step could be selected by having every node flip a coin with probability q of being selected. Alternatively, we could choose n random integers with replacement in the range $[1 \ldots n]$ and all nodes whose number is chosen are selected, or a single random integer k in the range $[1 \ldots n]$ could be chosen and the $\frac{n}{2}$ nodes $k, k+1, \ldots k+\frac{n}{2}-1$ (wrapping around to 1 if node IDs greater than n are included) are selected.) The following lemma, first proved by Luby [Lub88] in a slightly different form, describes the expected number of steps to color all the nodes.

LEMMA A.1 (random circuit lemma). *Consider a directed acyclic graph, G, of depth d with P paths from the inputs to the outputs. If in each step, any node which has all its predecessors colored has at least a constant probability $q > 0$ of being colored in this step, then the expected number of steps to color all the output nodes is at most $(6/q)(d + \log P)$, and the probability that more than $(5c/q)(d + \log P)$ steps are used is less than $1/P^c$.*

Proof. We analyze the expected number of steps using an auxiliary graph G_1. Each path from an input to an output in G is an independent chain in G_1. (See Fig. 9.) Thus each node in G_1 is associated with a unique node in G, while each node in G is associated with a set of nodes in G_1.

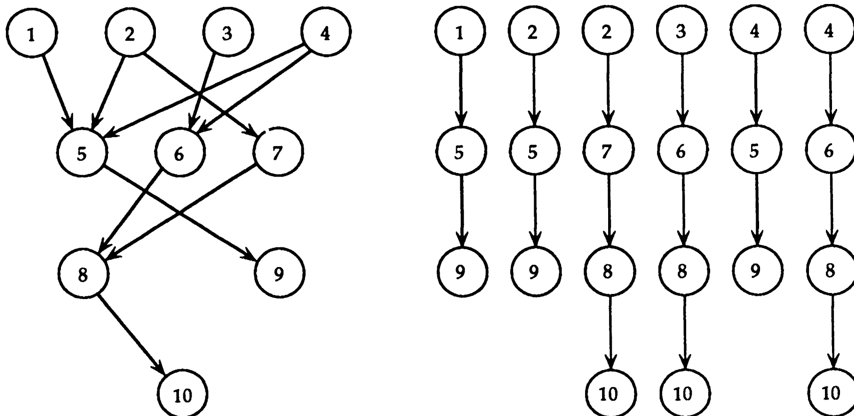


FIG. 9. *Example graph.*

Suppose that we color both G and G_1 simultaneously. In each step we choose a random set of nodes in G, and for each node v chosen in G, we choose all the nodes in G_1 that are associated with v. Let t_v be the step in which node v in G is colored, and let T_v be the earliest step that all the nodes associated with v in G_1 are colored. It is easy to prove by induction that for any sequence of selections and for any node v, $t_v = T_v$. Thus the final output of G is colored at exactly the same step as the final node in G_1 is colored.

We now bound the expected number of steps to color G_1. We will assume that all chains are of length exactly d, which can only increase the expected number of steps. For a single chain, the probability that the next entry in the chain is selected in the current

step is $\geq q$. Thus the expected number of steps to color the chain is just the expected number of steps to get d heads when flipping a coin with probability $\geq q$ of heads. Thus the expected number of steps is $\leq d/q$.

To calculate the probability that a chain is not done after kd/q steps (for some constant $k > 1$), we use the Chernoff bound on the tail of the binomial distribution. Let us focus our attention on a particular chain. First, consider the case where $d > \log P$.

Let $X_i = 1$ if the chain is selected in the ith step and 0 otherwise.
Let $m = kd/q$ be the number of steps. Let $P[X_i = 1] = q$.
Note that q is actually a lower bound on $P[X_i = 1]$, but we set $P[X_i = 1] = q$ since doing so simplifies the analysis without detracting from its correctness.
Let the number of steps in which the chain is selected $= S_m = \sum_{l=1}^{l=m} X_i$.
A chain is completed if it is selected in at least d steps.
The probability that the chain is uncompleted after m steps $= q_f = P[S_m < d]$.

Casting the above in the form of $P[S_m \leq (1-\epsilon)mq] \leq 2^{(-\epsilon^2 mq/2\ln 2)}$ from Lemma A.5, gives us

$$\epsilon = 1 - \frac{1}{k} \Rightarrow q_f \leq 2^{\left(-\frac{(1-\frac{1}{k})^2 mq}{2\ln 2}\right)} = 2^{(-2d)\frac{(1-\frac{1}{k})^2 k}{4\ln 2}} \leq \frac{1}{2^{2d}},$$

provided that

$$\frac{(1-\frac{1}{k})^2 k}{4\ln 2} \geq 1.$$

Choosing $k = 5$, gives

$$\frac{(1-\frac{1}{k})^2 k}{4\ln 2} = \frac{4}{5\ln 2} \geq 1.$$

Since $d \geq \log P$, $q_f \leq 1/2^{2d} \leq 1/P^2$.

Hence, the probability that a particular path is uncompleted after $5d/q$ steps is less than $1/P^2$. Hence, the probability that there is at least one uncompleted path $= P[\cup\, i$ is uncompleted$] \leq \sum_{i=1}^{i=P} P[i$ is uncompleted$] < 1/P$. Since each block of $5d/q$ steps has at least a $1 - (1/P)$ probability of completing all chains, independent of other blocks, the probability that there is an incomplete chain after $(5cd/q)$ steps is less than $\frac{1}{P^c}$.

For any integer constant $l > 1$, the probability that there is at least one incomplete chain after lm steps is $< P/P^{2l}$. Let T be the number of steps until all chains are completed. It follows that $E[T] < m + 2m/P^3 + 3m/P^5 + \ldots < 6d/q$ for $P \geq 3$.

If $d < \log P$, the analysis holds by choosing $m = 5\log P/q$ in the above analysis. This completes the proof of the expected number of steps to color G_1.

Since we showed earlier that the number of steps to color all chains is also the number of steps to color G, the lemma follows. □

We conclude this appendix by presenting a few lemmas which we use in this paper.

LEMMA A.2. *Let X be a random variable such that $X \leq m$ and $E[X] \geq \alpha m$ for some constant $0 < \alpha < 1$. Then, $P[X \geq \alpha m/2] > \frac{\alpha}{2}$.*

Proof. We shall show that if $P[X \geq \alpha m/2] \leq \alpha/2$, then $E[X] < \alpha m$, which is a contradiction. Without loss of generality, let $P[X \geq \alpha m/2] = \alpha/2$, since using a lower value for α will only further lower the value for $E[X]$.

$$E[X] \leq \frac{\alpha}{2} \times m + \left(1 - \frac{\alpha}{2}\right) \times \frac{\alpha m}{2} = \alpha m - \frac{\alpha^2 m}{4}.$$

But, by assumption, $E[X] = \alpha m$. Therefore, $P[X \geq \alpha m/2] > \frac{\alpha}{2}$. □

LEMMA A.3. *Consider the coupon collector problem in which a selection consists of choosing uniformly and randomly from a set of n coupons. Define X = number of selections until all coupons have been selected. Then, $E[X] = n \ln n$ and $P[X \geq 2n \ln n] \leq \frac{1}{n}$.*

LEMMA A.4. *Define m identically distributed random variables $\{X_i\}$, where $P[X_i = 1] \geq p$ and $P[X_i = 0] \leq 1 - p$. (Note that the X_i need not be independent for the following to hold.) Define $X = \sum_{i=1}^{i=m} X_i$. Then, $E[X] \geq mp$. Also, using Lemma* A.2, $P[X \geq mp/2] \geq \frac{p}{2}$.

LEMMA A.5 [Che52]. *Let X_i be independent and identically distributed random variables such that $P[X_i = 1] = p$ and $P[x_i = 0] = 1 - p$. Define $S_n = \sum_{i=1}^{i=n} X_i$. Then, for any $\epsilon < 1$,*

$$P[S_n \leq (1-\epsilon)np] \leq e^{\frac{-\epsilon^2 np}{2}} = 2^{\frac{-\epsilon^2 np}{2 \ln 2}}.$$

Remark. In our settings, the X_i are not independently and identically distributed random variables. However, the probability that a given X_i is 1 is less than p, independent of the other X_j, so the above bounds continue to hold.

LEMMA A.6. *Let X be a positive random variable. Then $P[X > a] < E[X]/a$ for any $a > 0$.*

LEMMA A.7 [Fel57]. *If m selections are made with replacement from m items, the probability that a given item is not selected is $(1-(1/m))^m$. $\forall m \geq 2, \frac{1}{4} \leq (1-(1/m))^m \leq 1/e$.*

LEMMA A.8. *If $2s$ random selections are made with replacement from s distinct items, then P[less than $s/2$ distinct items are selected] $\leq \frac{1}{2}^s \leq \frac{1}{2}$.*

Proof. We can consider each selection of $2s$ items to be a vector of length $2s$. Hence, there are s^{2s} such different vectors. The number of vectors with less than $\frac{s}{2}$ distinct elements is at most

$$\binom{s}{\frac{s}{2}} \left(\frac{s}{2}\right)^{2s} < 2^s \left(\frac{s}{2}\right)^{2s} = \frac{s^{2s}}{2^s}.$$

Therefore, the probability that less than $s/2$ distinct items are selected $< s^{2s}/2^s s^{2s} = \frac{1}{2^s}$. □

Acknowledgments. We would like to thank Mike Luby for useful discussions and for showing us his version of Lemma 3.1.

REFERENCES

[AH90] J. ASPNES AND M. HERLIHY, *Wait-free data structures in the asynchronous pram model*, in Proceedings of the 2nd Symposium on Parallel Architectures and Algorithms, Crete, Greece, 1990, pp. 340–349.

[AHU74] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Reading, MA, 1974.

[AKS83] M. AJTAI, J. KOMLOS, AND E. SZEMEREDI, *An $O(n \log n)$ sorting network*, in Proceedings of the 15th ACM Symposium on Theory of Computing, Boston, MA, 1983, pp. 1–9.

[ALS90] H. ATTIYA, N. LYNCH, AND N. SHAVIT, *Are wait-free algorithms fast?* in Proceedings of the 31st IEEE Symposium on Foundations of Computer Science, St. Louis, MO, 1990, pp. 55–64.

[AM89] R. ANDERSON AND G. MILLER, *Deterministic parallel list ranking*, in Lecture Notes in Computer Science, 319, J. Reif, ed., Springer-Verlag, New York, 1989; first appeared in Aegean Workshop on Computing '88.

[AW91] R. ANDERSON AND H. WOLL, *Wait-free parallel algorithms for the union-find problem*, in Proceedings of the 23rd Annual ACM Symposium on Theory of Computing, New Orleans, LA, 1991, pp. 370–380.

[BMP91] J. BECKER, C. MARTEL, AND A. PARK, *General asynchrony is not expensive for* PRAMs, in Proceedings of the 5th International Parallel Processing Symposium, Anaheim, CA, 1991, pp. 70–75.

[BR90] J. BUSS AND P. RAGDE, *Certified write-all on a strongly asynchronous pram*, Tech. Report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1990, manuscript.

[Bre74] R. P. BRENT, *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach., 21 (1974), pp. 201–206.

[Che52] H. CHERNOFF, *A measure of asymptotic efficiency for tests of hypothesis based on the sums of observations*, Ann. Math. Statist., 23 (1952), pp. 493–507.

[CV89] R. COLE AND U. VISHKIN, *Optimal parallel algorithms for expression tree evaluation and list ranking*, in Lecture Notes in Computer Science, 319, J. Reif, ed., Springer-Verlag, New York, 1989; first appeared in Aegean Workshop on Computing '88.

[CZ89] R. COLE AND O. ZAJICEK, *The* APRAM: *Incorporating asynchrony into the* PRAM *model*, in Proceedings of the Symposium on Parallel Architectures and Algorithms, Santa Fe, NM, 1989, pp. 169–178.

[CZ90a] ———, *The* APRAM: *A model for asynchronous parallel computation*, Tech. Report, New York University, New York, NY, 1990.

[CZ90b] ———, *The expected advantage of asynchrony*, in Proceedings of the Second Symposium on Parallel Algorithms and Architectures, Crete, Greece, 1990, pp. 85–94.

[Fel57] W. FELLER, *An Introduction to Probability Theory and Its Applications*, John Wiley, New York, 1957.

[Gib89] P. B. GIBBONS, *A more practical* PRAM *model*, in Proceedings of the Symposium on Parallel Architectures and Algorithms, Santa Fe, NM, 1989, pp. 158–168.

[Her88] M. HERLIHY, *Impossibility and universality results for wait-free synchronization*, in Proceedings of the 7th Symposium on Principles of Distributed Computing, 1988, pp. 276–290; Also in Trans. Programming Languages and Operating Systems, 31(1991), pp. 123–149.

[Joh89] B. W. JOHNSON, *Design and Analysis of Fault Tolerant Digital Systems*, Addison Wesley, Reading, MA, 1989.

[KPS90] Z. M. KEDEM, K. V. PALEM, AND P. G. SPIRAKIS, *Efficient robust parallel computations*, in Proceedings of the 22nd Annual Symposium on Theory of Computing, Baltimore, MD, 1990, pp. 138–148.

[KR90] R. M. KARP AND V. RAMACHANDRAN, *A survey of parallel algorithms for shared memory machines*, in Theoret. Comput. Sci., North Holland, Amsterdam, 1990.

[KRS90] C. KRUSKAL, L. RUDOLPH, AND M. SNIR, *A complexity theory of efficient parallel algorithms*, Theoret. Comput. Sci., 71 (1990), pp. 95–132.

[KS89] P. KANELLAKIS AND A. SHVARTSMAN, *Efficient parallel algorithms can be made robust*, in Proceedings of the 8th Symposium on Principles of Distributed Computing, Edmonton, Alberta, Canada, 1989, pp. 211–221.

[KS91] ———, *Efficient parallel algorithms on restartable fail-stop processors*, in Proceedings of the 10th Symposium on Principles of Distributed Computing, Montreal, Canada, 1991.

[Lub88] M. LUBY, *On the parallel complexity of symmetric connection networks*, Tech. Report 214/88, University of Toronto, Department of Computer Science, Toronto, Ontario, Canada, 1988.

[MPS89] C. U. MARTEL, A. PARK, AND R. SUBRAMONIAN, *Optimal asynchronous algorithms for shared memory parallel computers*, Tech. Report CSE 89-8, Division of Computer Science, University of California, Davis, CA, 1989.

[MR85] G. MILLER AND J. REIF, *Parallel tree contraction and its applications*, in Proceedings of the 26th IEEE Symposium on Foundations of Computer Science, Portland, OR, 1985, pp. 478–489.

[MS90] C. U. MARTEL AND R. SUBRAMONIAN, *Asynchronous algorithms for list ranking and transitive closure*, in Proceedings of the International Conference on Parallel Processing, St. Charles, IL, 1990, pp. 60–63.

[MSP90] C. U. MARTEL, R. SUBRAMONIAN, AND A. PARK, *Asynchronous* PRAMs *are (almost) as good as synchronous* PRAMs, in Proceedings of the 31st IEEE Symposium on Foundations of Computer Science, St. Louis, MO, 1990, pp. 590–599.

[Nis90] N. NISHIMURA, *Asynchronous shared memory parallel computation*, in Proceedings of the 2nd Symposium on Parallel Algorithms and Architectures, Crete, Greece, 1990, pp. 76–84.

[Shv89] A. SHVARTSMAN, *Achieving optimal* CRCW PRAM *fault tolerance*, Tech. Report CS-89-49, Department of Computer Science, Brown University, Providence, RI, December 1989.

[SM90] R. SUBRAMONIAN AND C. U. MARTEL, *How to emulate synchrony*, Tech. Report CSE 90-26, Department of Computer Science, University of California, Davis, CA, 1990.

[SS83] R. D. SCHLICHTING AND F. B. SCHNEIDER, *Fail-stop processors: An approach to designing fault-tolerant computing systems*, ACM Trans. Comput. Systems, 1 (1983), pp. 222–238.

[SV84] L. STOCKMEYER AND U. VISHKIN, *Simulation of random access machines by circuits*, SIAM J. Comput., 13 (1984), pp. 409–421.

[Val90] L. VALIANT, *A bridging model for parallel computation*, CACM, 33 (1990), pp. 103–111.

[Vis84] U. VISHKIN, *A parallel-design distributed implementation* (PDDI) *general purpose parallel computer*, Theoret. Comput. Sci., 32 (1984), pp. 157–172.

[Wyl81] J. C. WYLLIE, *The Complexity of Parallel Computation*, Ph.D. thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1981.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1100–1111, December 1992
© 1992 Society for Industrial and Applied Mathematics
006

POLYNOMIAL TIME PRODUCTIVITY, APPROXIMATIONS, AND LEVELABILITY*

JIE WANG†

Abstract. This paper studies polynomial-time approximations to intractable sets using the concept of p-productivity. It is shown that every (deterministic and nondeterministic) superpolynomial-time computable p-productive set is p-levelable. All $\leq_m^p$-complete sets for any deterministic superpolynomial time class are shown to be p-productive. It is then shown that the complement of any honest k-creative set in NP is p-levelable. This settles an open problem in Homer [*Theoret. Comput. Sci.*, 47 (1986), pp. 169–180].

Key words. p-productivity, p-approximations, p-levelability

AMS(MOS) subject classification. 68Q15

1. Introduction. This paper studies polynomial-time approximations to sets not in P, using the concept of p-productivity. P-productive sets were first defined and investigated by Ko and Moore [7] in complexity theory as part of the study of polynomial-time approximations to intractable sets, and they showed that p-productive sets cannot belong to the complexity class DTIME(2^{poly}) under their definition. The idea of p-productivity is to provide a mechanism that witnesses intractability of a set in polynomial time. In general, a set A is p-productive for P if there is a total polynomial-time computable function f, which provides an effective witness that A is not in P in the sense that, whenever a Turing machine M_i witnesses a language in P, $L(M_i) \subseteq A \Rightarrow f(i) \in A - L(M_i)$. Wang [14], [16] realized that definitions of p-productivity depend on the indexing, and that with a slow indexing of the functions it is possible to have p-productive sets for P in E, where E = DTIME($2^{O(n)}$). This is because with a slow indexing there is a universal predicate for polynomial-time functions within E. We show that this property holds in any superpolynomial time class as well. This allows us to study p-productivity within these classes. We then use the concept of p-productivity to study polynomial-time approximations to (deterministic and nondeterministic) exponential time and superpolynomial time computable sets.

An often-studied type of approximation for sets not in P was introduced by Meyer and Paterson [9]. A set $A \notin$ P has an approximation algorithm $\mathcal{A}$ if it halts in polynomial time and on each input x it outputs either 1 (accept), -1 (reject), or ? (does not know the answer), where $\mathcal{A}(x) = 1$ implies $x \in A$ and $\mathcal{A}(x) = -1$ implies $x \notin A$. An approximation algorithm is *optimal* if no other polynomial-time algorithm can correctly decide infinitely many more inputs. A notion that describes a set having an approximation algorithm but no optimal one is *p-levelability* [6]. A set A not in P is p-levelable if any algorithm for A can be sped up by a polynomial infinitely often. More precisely, given any algorithm M for A and polynomial p, another algorithm M' can always be found for A such that M' accepts infinitely many inputs $x \in A$ in polynomial time on which M exceeds $p(|x|)$ many steps. A p-levelable set contains infinite subsets in P but it does not contain a largest P subset. A set A is bi-p-levelable if both A and $\overline{A}$ are p-levelable. Clearly, bi-p-levelability describes two-sided polynomial-time approximations to sets not in P.

*Received by the editors January 16, 1991; accepted for publication (in revised form) December 2, 1991. This research was supported in part by National Science Foundation grant CCR-9108899. Part of this research was performed while the author was in Boston University supported by National Science Foundation grant CCR-8814339 and a Boston University Presidential Fellowship.

†Mathematics and Computer Science Department, Wilkes University, Wilkes-Barre, Pennsylvania 18766.

P-productivity is a natural and powerful concept to study p-levelability. Let T be a time constructible function, which dominates any polynomial. We prove that every p-productive set for P in DTIME($T(n)$) is p-levelable, and every $\leq_m^p$-complete set for DTIME($T(n)$) is p-productive for P. Since DTIME($T(n)$) is closed under complementation, the complement of every $\leq_m^p$-complete set for DTIME($T(n)$) is p-productive for P as well. Studying polynomial-time approximations to nondeterministic superpolynomial time computable sets is more difficult because the class of these sets is (probably) not closed under complementation. Whether every NE-complete set contains an infinite subset in P is a long-standing open problem [2], where NE = NTIME($2^{O(n)}$). We prove that the existence of p-productive sets for NP in NE is a sufficient condition to solve this problem. In particular, we prove that every p-productive set for NP in NTIME($T(n)$) is p-levelable. Under the assumption that p-productive sets for NP in NTIME($T(n)$) exist, we can prove that every $\leq_m^p$-complete set for NTIME($T(n)$) is p-levelable.

To study polynomial-time approximations to intractable problems in NP (assuming P $\neq$ NP), we consider the notion of k-creativity in NP. A set $A \in$ NP is *k-creative* if there is a total polynomial-time computable function f, which provides a witness that $\overline{A}$ is not in NTIME(n^k) in the sense that, whenever a Turing machine M_i witnesses a language in NTIME(n^k), $L(M_i) \subseteq \overline{A} \Rightarrow f(i) \in \overline{A} - L(M_i)$. If moreover, $f(i) \in A \Leftrightarrow f(i) \in L(M_i)$, then A is called *k-completely creative*. Every k-completely creative set is NP-complete [5]. Orponen, Russo, and Schöning [11] proved that every honest k-completely creative set is p-levelable. Homer [3] proved that the complement of every honest k-completely creative set in NP contains an infinite subset in NTIME(n^l) for some $l > 0$. He asked whether the complement of every honest k-completely creative set in NP contains an infinite subset in P. We affirmatively answer this question by showing a stronger result. We prove that the complement of any honest k-creative set in NP is p-levelable. Therefore, every honest k-completely creative set is bi-p-levelable.

2. Preliminaries. We assume familiarity with the standard polynomial-time many-one reductions. All of the problems considered here will be thought of as coded by finite strings over a fixed alphabet $\Sigma = \{0, 1\}$. As is customary, for a complexity class $\mathcal{C}$ we call a set A $\mathcal{C}$-complete if A is $\leq_m^p$-complete for $\mathcal{C}$. (Here $\leq_m^p$ denotes polynomial-time many-one reducibility.)

Let $\mathcal{N} = \{0, 1, \cdots\}$ be the set of all natural numbers. For each $x \in \mathcal{N}$, we use $str(x)$ to denote the binary representation of $x + 1$ with the leading 1 omitted. This is a bijection between $\mathcal{N}$ and the set of strings in Σ^*, and enable us to disregard the distinction between strings and representations of natural numbers. Languages are subsets of $\mathcal{N}$, or $\mathcal{N} \times \mathcal{N}$, etc. Functions are from $\mathcal{N}$ to $\mathcal{N}$, or $\mathcal{N} \times \mathcal{N}$ to $\mathcal{N}$, etc. For x and y in $\mathcal{N}$, write xy or $x \cdot y$ for the multiplication of x and y. For each $x \in \mathcal{N}$, we use $|x|$ to denote the length of the binary representation of x. So for all $x > 0$, $x < 2^{|x|} \leq 2x$. Regan [12] constructed a pairing function from $\Sigma^* \times \Sigma^*$ to Σ^* such that it is both linear-time computable and linear-time invertible. So using function str, it is easy to construct linear-time computable and invertible pairing function from $\mathcal{N} \times \mathcal{N}$ to $\mathcal{N}$. This enables us to disregard the distinction between pairs in $\mathcal{N} \times \mathcal{N}$ and natural numbers. Without loss of generality, we use $(\cdot, \cdot)$ to denote such a pairing function. If $f(\cdot, \cdot)$ is a function, we may interpret $f(\cdot, \cdot)$ as $f((\cdot, \cdot))$.

We use the standard deterministic/nondeterministic multitape Turing machines [4] as our computation model. All Turing machines can either accept languages or compute functions. A program (index) i is an integer, which simply codes up the states, symbols, tuples, etc. of the ith Turing machine M_i. Let $M_0, M_1, \cdots$ be a fixed enumeration of all (deterministic and nondeterministic) Turing machines. For convenience, denote by

DTM a deterministic Turing machine and TM a Turing machine. Let $L_i = L(M_i) = \{x : M_i \text{ accepts } x\}$ for all i.

For each set A not in P, $\mathcal{A}$ is a polynomial-time approximation (in short, p-approximation) algorithm for A if:

1. There exists a polynomial p such that for any x $\mathcal{A}(x)$ halts within $p(|x|)$ steps and $\mathcal{A}(x) = 1$ (accepts x), -1 (rejects x), or ? (does not know the answer).
2. For every x, $\mathcal{A}(x) = 1 \Rightarrow x \in A$, and $\mathcal{A}(x) = -1 \Rightarrow x \notin A$.
3. $\{x : \mathcal{A}(x) \neq ?\}$ is infinite.

A p-approximation algorithm $\mathcal{A}$ for A is *optimal* if there are no p-approximation algorithms for $A - \{x : \mathcal{A}(x) \neq ?\}$. A p-approximation $\mathcal{A}$ is a one-sided approximation for A if $\{x : \mathcal{A}(x) = 1\}$ is infinite. A notion that describes a set having a one-sided approximation algorithm, but no optimal one, is p-levelability defined in [6], [11].

DEFINITION 2.1. Given a set A, a DTM M, and a function f on the natural numbers, denote by $E(M, f)$ the set $\{x \in A : M \text{ accepts } x \text{ in time } f(|x|)\}$. A set A is p-levelable if, given any recognizer M for A and polynomial p, it is possible to find another recognizer M' for A and polynomial p' such that the difference $E(M', p') - E(M, p)$ is infinite.

Let A and B be two sets. $B \in$ P is a largest P subset of A if $A - B$ does not contain any infinite subsets in P. So p-levelability describes the property of a set that contains an infinite subset in P, but no largest one. A set A is bi-p-levelable if both A and $\overline{A}$ are p-levelable. Clearly, bi-p-levelability describes a two-sided p-approximation for sets not in P.

A polynomial-time computable function f is polynomially honest (in short, p-honest) if there is a polynomial p such that $p(|f(x)|) \geq |x|$ whenever $f(x)$ is defined. A function f is length increasing if $|f(x)| > |x|$ whenever $f(x)$ is defined. Given any number r, we use $\lfloor r \rfloor$ to denote the largest integer $\leq r$.

A function $T(n)$ is time constructible if $T(n)$ can be computed by a DTM in $O(T(n))$ steps on every input of length n. Let $r_0, r_1, \cdots$ be some enumeration of recursive functions. This enumeration is uniformly time constructible if all the r_i's are time constructible and there is a DTM M such that on input (i, x) M outputs $r_i(|x|)$ in time $O(r_i(|x|))$. We also assume that $r_i(n) \geq n$ for all n. Let $T^k(n)$ denote $T(n)$ multiplied with itself k times. For any Turing machine M, denote by $T_M(x)$ the number of steps M takes on input x if it halts, and leave undefined otherwise. Without loss of generality, we assume that if $T_M(x)$ is defined then $T_M(x) \geq |x|$.

3. Approximations to superpolynomial-time sets. To present our idea about how to approach p-approximations to intractable sets using p-productivity, we first study p-approximations to E sets as an example. We then generalize this idea to any superpolynomial-time sets.

3.1. Approximations to E sets. We first define p-productive sets for P. To construct p-productive sets for P in E, Wang [16] considered those machines that witness languages in P in the following uniform way. Let $p_i(n) = n^{\sqrt[3]{|i|}} + i$. Define

$$\mathcal{PM} = \{M_i : M_i \text{ is a DTM and } (\forall x \in L(M_i))[M_i \text{ accepts } x \text{ within } p_i(|x|) \text{ steps}]\}.$$

Clearly, P $= \{L(M_i) : M_i \in \mathcal{PM}\}$. The following definition is from Wang [14], [16].

DEFINITION 3.1. A set A is *p-productive for* P (with respect to $\mathcal{PM}$) if there is a total polynomial-time computable function f, which is called a p-productive function, such that for all i, if $M_i \in \mathcal{PM}$, then $L_i \subseteq A \Rightarrow f(i) \in A - L_i$. A set A is *p-creative* for P if $\overline{A}$ is p-productive for P.

REMARK 3.2. One might argue that perhaps using time bound $\lambda n[n^{|i|} + |i|]$ and the enumeration $\{M_i : M_i$ is a clocked Turing machine and its time bound is $\leq n^{|i|} + |i|\}$ instead of the one we are using here would be more natural. However, this is not the case for two reasons. First, when constructing a p-creative set for P, one needs to consider a universal Turing machine, which simulates the computation of M_i on input x. In this case, both x and i become inputs to the universal Turing machine. Since $\lambda xi[|x|^{|i|}] = O(2^{|i| \log |x|})$, the language accepted by the universal Turing machine is not in DTIME($2^{O(|i|+|x|)}$). So using $\lambda xi[|x|^{|i|} + |i|]$ does not allow us to get p-creative sets for P in E. Second, the recursion theorem does not hold for clocked Turing machines [14]. The recursion theorem is powerful and plays an important role in our theory (e.g., the recursion theorem is used to prove that p-creativeness is equivalent to p-complete creativeness). The reader is referred to Wang [14], [16] for more details.

REMARK 3.3. As p-productive sets for P are defined under the enumeration $\mathcal{PM}$, we may call these sets *p-productive sets for* $\mathcal{PM}$. However, because the motivation of defining p-productive sets for P is to provide a mechanism that witnesses that a set is not in P, we feel that p-productive sets for P with respect to $\mathcal{PM}$ better addresses this motivation. In this paper, when there is no confusion on which enumeration we are working on, we use p-productive sets for P for simplicity.

Since p-productivity is indexing dependent, choosing a right indexing is important. In summary, we require that the indexing we choose meet the following three requirements.

1. There is a universal predicate for all polynomial-time computable functions in the time complexity class we consider (E in this section and DTIME($T(n)$) in the next section).
2. It is closed under the s-m-n theorem.[1] That is, let $q_0, q_1, \cdots$ be the polynomial-time bound used in the indexing, if M on input (x, y) halts in $q_{v(x)}(|y|)$ steps, where v is bounded by a polynomial, then by applying the s-m-n theorem there is a total, one to one, length-increasing, and polynomial-time computable function h such that $M_{h(x)}(y)$ halts in $q_{h(x)}(|y|)$ steps and $M_{h(x)}(y) = M(x, y)$.
3. It is closed under the double diagonalization construction. That is, there is an enumeration of all polynomial-time computable functions $f_0, f_1, \cdots$ such that the set $\{(i, x) : M_x$ is a DTM and accepts $f_i(i, x)$ within $q_x(|f_i(i, x)|)$ steps$\}$ is in the time complexity class we consider (E in this section and DTIME($T(n)$) in the next section.)

It was shown in [16] that the enumeration $\mathcal{PM}$ we defined above meets all these three requirements. It is straightforward to prove the following simulation lemma [16].

LEMMA 3.4. *$\{(i, x) : M_i$ is a* DTM *and accepts x* in $p_i(|x|)$ steps$\}$ *can be accepted by a two-tape* DTM *in time $O(|i|p_i^2(|x|))$.*

Consider the Kleene function defined by $K(i, x, n) = M_i(x)$, if M_i is a DTM and halts on input x within n steps, 0, otherwise. Let $f_i = \lambda x[K(i, x, p_i(|x|))]$, then $f_0, f_1, \cdots$ is an enumeration of all total polynomial-time computable functions. By Lemma 3.4, it is straightforward [16] to verify that the universal function $\lambda ix[f_i(x)]$ is computable in DTIME($2^{O(|i|+|x|)}$). Using the double diagonalization construction, Wang [16] proved that every $\leq_m^p$-hard set for E is p-productive for P. We now prove that every p-productive set for P in E is p-levelable.

THEOREM 3.5. *Every p-productive set for* P *in* E *is p-levelable.*

Proof. The proof consists of a series of lemmas. Each of these lemmas is interesting in its own right in terms of p-productivity.

[1] The version of s-m-n theorem in the polynomial setting can be found, for example, from [12], [14].

LEMMA 3.6. *Every p-productive set for* P *with a length-increasing p-productive function contains infinite subsets in* P.

Proof. Let A be a p-productive set for P with a length-increasing p-productive function h. Then

$$(1) \qquad (\forall i)[M_i \in \mathcal{PM} \Rightarrow [L_i \subseteq A \Rightarrow h(i) \in A - L_i]].$$

Let M_h be a DTM computing h within $O(n^l)$ steps on inputs of length n. Without loss of generality, assume that M_h is one-tape. Define a two-tape DTM M such that on input (x, y), if M_x is a DTM and accepts y within $p_x(|y|)$ steps or $M_h(x) = y$, then M accepts; otherwise, M rejects. From the Simulation Lemma 3.4 we have $T_M(x,y) \leq O(|x|(|y|^{\sqrt[3]{|x|}} + x + |x|^l)^2) \leq |y|^{O(\log|x| + \sqrt[3]{|x|})} + O(|x|(1 + x + |x|^l)^2)$. So by the s-m-n theorem, there is a total, length-increasing, and polynomial-time computable function g such that $M_{g(x)}(y) = M(x,y)$ and $T_{M_{g(x)}}(y) = T_M(x,y)$. We can pad g such that $T_{M_{g(x)}}(y) \leq p_{g(x)}(|y|)$. That is, $M_{g(x)} \in \mathcal{PM}$ for every x. From the construction we can see that if $M_x \in \mathcal{PM}$, then

$$(2) \qquad L_{g(x)} = L_x \cup \{h(x)\}.$$

Let i_0 be an index such that $L_{i_0} = \emptyset$ and $M_{i_0} \in \mathcal{PM}$. So $L_{i_0} \subseteq A$. Therefore, since $M_{i_0} \in \mathcal{PM}$, by (1), $h(i_0) \in A$. So, by (2), $L_{g(i_0)} = \{h(i_0)\} \subseteq A$. Since $M_{g(i_0)} \in \mathcal{PM}$, by (1), $hg(i_0) \in A - L_{g(i_0)}$, and so $hg(i_0) \neq h(i_0)$. Again, by (2), $L_{g^2(i_0)} = L_{g(i_0)} \cup \{hg(i_0)\} \subseteq A$ since $M_{g(i_0)} \in \mathcal{PM}$. We know that $M_{g^2(i_0)} \in \mathcal{PM}$, so by (1), $hg^2(i_0) \in A - L_{g^2(i_0)}$, and so $hg^2(i_0) \notin \{h(i_0), hg(i_0)\}$. Continuing this procedure, we will obtain the following set $B = \{h(i_0), hg(i_0), hg^2(i_0), \cdots\} \subseteq A$ such that $hg^i(i_0) \neq hg^j(i_0)$ when $i \neq j$. So B is infinite. That B is in P is obvious because h and g are both length-increasing polynomial-time computable. This completes the proof. □

LEMMA 3.7. *If $B \in$ P and $A \cup B$ is p-productive for* P, *then A is p-productive for* P. *Moreover, if $A \cup B$ has a length-increasing p-productive function, then so does A.*

Proof. Since $A \cup B$ is p-productive for P, there is a total, polynomial-time computable function f such that, for all i, if $M_i \in \mathcal{PM}$, then $L_i \subseteq A \cup B \Rightarrow f(i) \in (A \cup B) - L_i$. We want to show that there is a total, polynomial-time computable function g such that $(\forall i)[M_i \in \mathcal{PM} \Rightarrow [L_i \subseteq A \Rightarrow g(i) \in A - L_i]]$. Since $B \in$ P, there is a total, length-increasing, and polynomial-time computable function h such that $L_{h(i)} = L_i \cup B$, and $M_i \in \mathcal{PM}$ implies $M_{h(i)} \in \mathcal{PM}$ by padding h. Let $g(i) = f(i)$ if $f(i) \notin B$, and $fh(i)$ otherwise. Clearly, g is total, polynomial-time computable and if f is length increasing, then so is g. Given i, if $M_i \in \mathcal{PM}$ and $L_i \subseteq A$, then $L_i \subseteq A \cup B$. If $f(i) \notin B$, then $f(i) \in A - L_i$, namely, $g(i) \in A - L_i$. If $f(i) \in B$, then consider $L_i \cup B$. Since $L_i \subseteq A$, $L_i \cup B \subseteq A \cup B$. Namely, $L_{h(i)} \subseteq A \cup B$. So $fh(i) \in (A \cup B) - L_{h(i)}$. Hence, $fh(i) \in (A \cup B) - (L_i \cup B)$. Therefore, $g(i) = fh(i) \in A - L_i$. This completes the proof. □

LEMMA 3.8. *If A is p-productive for* P *with a length-increasing p-productive function, then A is p-levelable.*

Proof. Let A be as described above. Suppose that A is not p-levelable. Then there is an optimal one-sided p-approximation algorithm $\mathcal{A}$ for the positive site of A. Let $B = \{x : \mathcal{A}(x) = 1\}$, then by definition, $B \in$ P, $B \subseteq A$, and there are no one-sided p-approximation algorithms for the positive site of $A - B$. However, since $A = (A-B) \cup B$, by Lemma 3.7, $A - B$ is p-productive for P with a length-increasing p-productive function. By Lemma 3.6, $A - B$ contains an infinite subset in P. This is a contradiction. □

LEMMA 3.9. *Every p-productive set for* P *in* E *has a length-increasing p-productive function.*

Proof. Suppose A is p-productive for P in E, then there is a total, polynomial-time computable function f such that $(\forall i)[M_i \in \mathcal{PM} \Rightarrow [L_i \subseteq A \Rightarrow f(i) \in A - L_i]]$. For all i and x, let

$$M(i,x) = \begin{cases} M_i(x), & \text{if } |x| > |i|, \\ \text{“accept,”} & \text{if } |x| \leq |i| \text{ and } x \in A, \\ \text{“reject,”} & \text{otherwise.} \end{cases}$$

there is a total, polynomial-time computable function h such that $M_{h(i)}(x) = M(i,x)$ with the same time complexity, i.e., $T_{M_{h(i)}}(x) = T_M(i,x)$. From the construction, it is clear that $T_M(i,x) = O(|i|T^2_{M_i}(x)) + 2^{c|i|}$ for some constant $c > 0$ since $A \in$ E. So if $M_i \in \mathcal{PM}$, then $T_M(i,x) = c_1|i|(|x|^{\sqrt[3]{|i|}} + i)^2 + 2^{c|i|}$ for some $c_1 > 0$. By suitably padding h such that $h(i) > c_1|i|(1+i)^2 + i^c$ and $|h(i)| \geq (3\sqrt[3]{|i|} + \log|i| + |i|)^3$ we have $T_{M_{h(i)}}(x) \leq |x|^{\sqrt[3]{|h(i)|}} + h(i)$. Hence, $M_i \in \mathcal{PM}$ implies $M_{h(i)} \in \mathcal{PM}$.

Let i be an index such that $M_i \in \mathcal{PM}$ and $L_i \subseteq A$. From the construction, we know that $L_i \subseteq A$ implies $L_{h(i)} \subseteq A$. Now we know $M_i \in \mathcal{PM}$ implies $M_{h(i)} \in \mathcal{PM}$. So by the definition of p-productive sets, $L_i \subseteq A \Rightarrow L_{h(i)} \subseteq A \Rightarrow fh(i) \in A - L_{h(i)}$. We claim that for such i, $|fh(i)| > |i|$. If it were not true, then by the construction of the machine $M_{h(i)}$, $fh(i)$ is accepted by $M_{h(i)}$, because $fh(i) \in A$. Namely, $fh(i) \in L_{h(i)}$. This contradicts the p-productivity. Thus, $|fh(i)| > |i|$. So, from the construction, $fh(i) \in L_i$ if and only if $fh(i) \in L_{h(i)}$. Hence, $fh(i) \notin L_{h(i)}$ implies $fh(i) \notin L_i$. Therefore, $(\forall i)[M_i \in \mathcal{PM} \Rightarrow [L_i \subseteq A \Rightarrow |fh(i)| > |i| \text{ and } fh(i) \in A - L_i]]$.

Now for all i, let $g(i) = fh(i)$ if $|fh(i)| > |i|$, and $i1$ otherwise, where $i1$ denotes the concatenation of i with 1. Then g is total, length increasing, and polynomial-time computable. Clearly, A is p-productive for P with productive function g. □

From Lemmas 3.9 and 3.8, we know that every p-productive set for P in E is p-levelable. This completes the proof of Theorem 3.5. □

REMARK 3.10. Lemmas 3.6, 3.7, and 3.8 also hold if we replace “length-increasing” for “p-honest” in their statements.

Since E is closed under complementation, it is easy to see the following corollary.

COROLLARY 3.11. (1) *Every E-complete set is bi-p-levelable* [11]. (2) *Every p-creative set for* P *in* E *has a length-increasing p-productive function.*

3.2. Approximations to DTIME($T(n)$) sets. We now generalize the idea in §3.1 to deterministic superpolynomial time class DTIME($T(n)$), where T dominates any polynomial. We can see from §3.1 that if there is an enumeration within DTIME($T(n)$) that meets the three requirements we mentioned there, then all the results we proved in §3.1 can be generalized to DTIME($T(n)$). So our task is to construct such an enumeration. Readers may skip most of this section if they believe such an enumeration exists and read Theorems 3.17 and 3.19 directly.

To define a function T that dominates any polynomial for time complexity classes nicely, T should be time constructible, monotonic, and greater than any polynomial. Moreover, a technical assumption is needed in our proof. That is, for all k and all n, $T^k(n)$ is not greater than $T(n^k)$. This assumption was first found in the proof of the main theorem of [2]. In his proof Berman used an assumption that $\lfloor T^{-1}(\lfloor g^{-1}(T(n))\rfloor)\rfloor \geq \lfloor g^{-1}(n)\rfloor$, where g is a monotonically increasing function. Taking $g(n) = n^k$ we get our assumption here. We shall see that this is a reasonable assumption because all the natural superpolynomial-time bounds satisfy this assumption.

DEFINITION 3.12. A function T *dominates* any polynomial if T is time constructible and satisfies the following three conditions.[2]

1. $(\forall j)[\lim_{n\to\infty} n^j/T(n) = 0]$.
2. T is monotonic, i.e., $(\forall n)[T(n+1) \geq T(n)]$.
3. $(\forall k)(\forall n)[T^k(n) \leq T(n^k)]$.

Now we shall show that the following natural superpolynomial-time bounds 2^{cn}, $2^{c\sqrt[l]{n}}$, $n^{c\log n}$, $n^{c\log^{(l)} n}$, $n^{c\log^* n}$, etc. satisfy Definition 3.12. Recall that $\log^* n = \min\{i : \log^{(i)} n \leq 1\}$, where $\log^{(l)} n = \log(\log^{(l-1)} n)$, $\log^{(0)} n = n$.

PROPOSITION 3.13. *Functions* $2^{2^{cn}}$, 2^{n^l+l}, 2^{cn}, $2^{c\sqrt[l]{n}}$, $n^{c\log n}$, $n^{c\log^{(l)} n}$, $n^{c\log^* n}$, *etc. dominate any polynomial, where l is a positive integer and c is a positive number.*

Proof. It suffices to show that if $U(n) = n^{\mu(n)}$ is time-constructible such that $\mu(n)$ is monotonic and $\lim_{n\to\infty}\mu(n) = \infty$, then U dominates any polynomial. Conditions 1 and 2 are easily seen satisfied. We present a proof for Condition 3. For any $k > 0$, $U^k(n) = n^{k\mu(n)} \leq n^{k\mu(n^k)} = (n^k)^{\mu(n^k)} = U(n^k)$. This completes the proof. □

From now on, we shall fix the use of T to denote a function that is time constructible and dominates any polynomial. We now present an enumeration within DTIME$(T(n))$ that meets the three requirements we mentioned in §3.1.

Since $T(n) = n^{\log T(n)/\log n}$, for simplicity, we assume that $\log T(n)/\log n$ is monotonic. From Proposition 3.13 we can see that this is a reasonable assumption. This assumption greatly simplifies our indexing construction, although we can construct an enumeration without this assumption [15]. Let $\sigma(n) = \log T(\lfloor\sqrt{n}\rfloor)/\log n$, and $\gamma(n) = \lfloor\sqrt{\lfloor\sigma(n)\rfloor}/2\rfloor$. For simplicity, we will eliminate the floors when there is no confusion.

LEMMA 3.14. (1) $T(\sqrt{n})$ *dominates any polynomial.* (2) σ *is monotonic and* $\lim_{n\to\infty}\sigma(n) = \infty$. (3) $\gamma(n)$ *is computable in* DTIME$(T(\sqrt{n}))$.

Proof. That $T(\sqrt{n})$ is time constructible is obvious since $\lfloor\sqrt{n}\rfloor$ is the largest integer m such that $m^2 \leq n$, and so it can be computed by a DTM in time $O(n|m|^2 + T(m)) \leq O(T(\sqrt{n}))$ on any input of length n. It is easy to verify that $T(\sqrt{n})$ satisfies the three conditions in Definition 3.12 and that Statement 2 is true.

We now prove Statement 3. Clearly, $\lfloor\sigma(n)\rfloor$ is the largest integer k such that $n^k \leq T(\sqrt{n})$ since $n^{\sigma(n)} = T(\sqrt{n})$. The following simple algorithm finds this k on input n. Let $z = T(\sqrt{n})$, and set $k = 0$. Loop: if $n^k \leq z$ and $n^{k+1} > z$ then stop and output k; otherwise increase k by 1. Obviously, the algorithm can be carried out deterministically in time $O(T(\sqrt{n}) + k|n|^{k+1})$. By the algorithm, $n^k \leq T(\sqrt{n})$, so $k \leq \sigma(n)$, $n \leq T^{1/k}(\sqrt{n})$, and $|n|^{k+1} = (O(\log n))^{k+1} \leq O(\log^{k+1} T(\sqrt{n}))$. Hence, the algorithm can be carried out in time $O(T(\sqrt{n}))$. To compute $\gamma(n)$ we find the largest number m such that $m^2 \leq \sigma(n)/4$ and $(m+1)^2 > \sigma(n)/4$. This can be carried out in deterministic time $O(m|m|^2 + T(\sqrt{n})) \leq O(\sigma(n) + T(\sqrt{n})) = O(T(\sqrt{n}))$. □

Let $t_i = \lambda n[n^{\gamma(|i|)} + T(\sqrt[3]{|i|})]$, and $g_i = \lambda x[K(i, x, t_i(|x|))]$, where $K(\cdot,\cdot,\cdot)$ is the Kleene function we defined in §3.1, then $g_0, g_1, \cdots$ is an enumeration of all total polynomial-time computable functions. It is clear that $t_0, t_1, \cdots$ is uniformly time constructible, so it is straightforward to prove that $\{(i, x) : M_i$ is a DTM and accepts x within $t_i(|x|)$ steps$\}$ can be accepted by a two-tape DTM in time $O(|i|t_i^2(|x|))$, as in [14].

THEOREM 3.15. *The universal function* $\lambda ix[g_i(x)]$ *is computable in* DTIME$(T(n))$.

Proof. It is clear that σ is monotonic, and so is γ. Hence, for any i and x, $(|i| + |x|)^{\sigma(|i|)} \leq (|i| + |x|)^{\sigma(|i|+|x|)} = T(\sqrt{|i| + |x|})$. Clearly, the universal function $\lambda ix[g_i(x)]$ is computable deterministically in time $O(|i|t_i^2(|x|))$. This is less than or equal to

[2]Condition 2 can be weakened to $(\forall^\infty n)[T(n+1) \geq T(n)]$, and Condition 3 can be weakened to $(\forall^\infty n)(\forall^\infty k)[T^k(n) \leq T(n^k)]$.

$O(|i||x|^{2\gamma(|i|)} + |i||x|^{\gamma(|i|)}T(\sqrt[3]{|i|}) + |i|T(\sqrt[3]{|i|^2}))$
$\leq O((|i| + |x|)^{2\gamma(|i|)+1} + (|i| + |x|)^{2\gamma(|i|)}T(\sqrt[3]{|i|}) + T(|i|))$
$\leq O((|i| + |x|)^{\sigma(|i|)}T(\sqrt[3]{|i|}) + T(|i|))$
$\leq O(T(\sqrt{|i| + |x|})T(\sqrt[3]{|i|}) + T(|i|))$
$\leq O(T(|i| + |x|))$.
That is, the universal function is computable in DTIME$(T(n))$. □

We will now define p-productive sets for P and NP using t_i. Let

$\mathcal{PM}_T = \{M_i : M_i$ is a DTM and $(\forall x \in L(M_i))[M_i$ accepts x within $t_i(|x|)$ steps$]\}$.

$\mathcal{NM}_T = \{M_i : (\forall x \in L(M_i))[M_i$ accepts x within $t_i(|x|)$ steps$]\}$.

Clearly $\mathrm{P} = \{L_i : M_i \in \mathcal{PM}_T\}$ and $\mathrm{NP} = \{L_i : M_i \in \mathcal{NM}_T\}$.

DEFINITION 3.16. A set A is *p-productive for* P (with respect to $\mathcal{PM}_T$) if there is a total polynomial-time computable function f, which is called a *p-productive function*, such that for all i, if $M_i \in \mathcal{PM}_T$, then $L_i \subseteq A \Rightarrow f(i) \in A - L_i$. A set A is p-creative for P (with respect to t_i) if $\overline{A}$ is p-productive for P.

We can similarly define p-productive sets for NP, and p-creative sets for NP with respect to $\mathcal{NM}_T$.

In what it follows, when we talk about p-creative sets and p-productive sets in DTIME$(T(n))$ we mean that they are defined with respect to $\mathcal{PM}_T$. The following theorem can be similarly obtained using the double diagonalization technique introduced in Wang [14], [16].

THEOREM 3.17. (1) *Every* $\leq_m^p$*-hard set for* DTIME$(T(n))$ *is p-creative for* P. (2) *Every* $\leq_m^p$*-hard set for* NTIME$(T(n))$ *is p-creative for* NP.

Proof. We present a proof for Statement 1. A proof for Statement 2 can be similarly obtained.

Define $Q_t = \{(i, x) : M_x$ is a DTM and accepts $g_i(i, x)$ within $t_x(|g_i(i, x)|)$ steps$\}$. By a straightforward calculation, we can verify that $Q_t \in$ DTIME$(T(n))$. This Q_t will force every $\leq_m^p$-hard set for DTIME$(T(n))$ to be p-creative for P. Let A be an arbitrary $\leq_m^p$-hard set for DTIME$(T(n))$. Then there is a total, polynomial-time computable function q such that $Q_t \leq_m^p A$ via q. Therefore, there is a j such that $q = g_j$. Let $f(x) = g_j(j, x)$, then f is a total, polynomial-time computable function. Now for all x, if $M_x \in \mathcal{PM}_T$, then $f(x) \in A \Leftrightarrow g_j(j, x) \in A \Leftrightarrow (j, x) \in Q_t$ (by reducibility) $\Leftrightarrow M_x$ accepts $g_j(j, x)$ within $t_x(|g_j(j, x)|)$ steps $\Leftrightarrow g_j(j, x) \in L(M_x) \Leftrightarrow f(x) \in L_x$. So it is easy to see that A is p-creative for P. □

Since DTIME$(T(n))$ is closed under complementation, we have the following corollary.

COROLLARY 3.18. *Every* $\leq_m^p$*-hard set for* DTIME$(T(n))$ *is p-productive for* P.

The reader can verify that the enumeration we defined above is also closed under the s-m-n theorem, so similar to the proof of Theorem 3.5, we can prove the following.

THEOREM 3.19. *Every p-productive set for* P *in* DTIME$(T(n))$ *is p-levelable.*

The proof consists of a series of lemmas, which are given in Lemma 3.20. Their proofs are omitted since the proof technique is the same as the one we used in Theorem 3.5. The reader may consult [15] for these proofs.

LEMMA 3.20. (1) *Every p-productive set for* P *with a p-honest p-productive function contains infinite subsets in* P. (2) *If* $B \in$ P *and* $A \cup B$ *is p-productive for* P *with a p-honest p-productive function, then* A *is p-productive for* P *with a p-honest p-productive function.* (3) *Every p-productive set for* P *with a p-honest p-productive function is p-levelable.* (4) *Every p-productive set for* P *in* DTIME$(T(n))$ *has a length-increasing p-productive function.*

As a direct corollary of Theorems 3.17 and 3.19 we know that every $\leq_m^p$-complete set for DTIME$(T(n))$ is bi-p-levelable. This result was first shown in [11] based on the main

theorem of [2] stating that every $\leq_m^p$-complete set for DTIME($T(n)$) is not P-immune and a result in [10] stating that any non-p-levelable set is the disjoint union of an infinite P-immune set and a set in P.

3.3. Approximations to NE sets. Whether every NE-complete set has an infinite P-subset is a long-standing open problem, which was raised in [2]. We show that the existence of p-productive sets for NP in NE is a sufficient condition to solve this problem. It is easy to prove that p-productive sets for NP exist in DTIME($2^{2^{\text{linear}}}$). It was shown in Wang [16] that all NE-complete sets are p-creative for NP. But since we do not know whether NE is closed under complementation, the existence of p-productive sets for NP in NE is open. Let $\mathcal{NM} = \{M_i : (\forall x \in L(M_i))[M_i \text{ accepts } x \text{ within } p_i(|x|) \text{ steps}]\}$. Clearly, NP $= \{L(M_i) : M_i \in \mathcal{NM}\}$. Consider the set $\{(i, x) : M_i \text{ is an NTM and accepts } x \text{ in } p_i(|x|) \text{ steps}\}$. It is easy to see that this set can be accepted by a two-tape, nondeterministic Turing machine in time $O(|i|p_i^2(|x|))$.

A set A is p-productive for NP (with respect to $\mathcal{NM}$) if there is a total polynomial-time computable function f, which is called a p-productive function, such that $(\forall i)[M_i \in \mathcal{NM} \Rightarrow [L_i \subseteq A \Rightarrow f(i) \in A - L_i]]$.

We first show that p-productivity for NP is closed under $\leq_m^p$-reducibility in the following sense.

THEOREM 3.21. *If $A \leq_m^p B$ and A is p-productive for* NP, *then B is p-productive for* NP.

Proof. Since A is p-productive for NP, there is a total polynomial-time computable function h such that $(\forall x)[M_x \in \mathcal{NM} \Rightarrow [L_x \subseteq A \Rightarrow h(x) \in A - L_x]]$. Let $A \leq_m^p B$ via f. So $f(y)$ can be computed deterministically in time $|y|^k + k$ for some k. By the s-m-n theorem, we can have a total, length-increasing, and polynomial-time computable function g such that $M_{g(x)}$ accepts y if $M_x(f(y))$ halts, is undefined otherwise, and $M_x \in \mathcal{NM}$ implies $M_{g(x)} \in \mathcal{NM}$ by suitably padding g. Since $z \in f(L_{g(x)}) \Rightarrow (\exists y \in L_{g(x)})[f(y) = z] \Rightarrow f(y) \in L_x \Rightarrow z \in L_x$, $f(L_{g(x)}) \subseteq L_x$. So for all x, if $M_x \in \mathcal{NM}$ then $L_x \subseteq B \Rightarrow f(L_{g(x)}) \subseteq B \Rightarrow L_{g(x)} \subseteq A$ (by reducibility) $\Rightarrow hg(x) \in A - L_{g(x)}$ (by p-productivity of A) $\Rightarrow fhg(x) \in B - L_x$ (by reducibility and the construction of $M_{g(x)}$). So B is p-productive for NP with p-productive function fhg. □

COROLLARY 3.22 (to the proof). *If $A \leq_m^p B$ and A is p-productive for* P, *then B is p-productive for* P.

Similar to the proofs of Lemmas 3.6, 3.7, 3.8, and 3.9, we can prove the following lemmas, which are given in Lemma 3.23. Their proofs are omitted.

LEMMA 3.23. (1) *Every p-productive set for* NP *with a length-increasing p-productive function contains infinite subset in* P. (2) *If $B \in$* P *and $A \cup B$ is p-productive for* NP *with a length-increasing p-productive function, then A is p-productive for* NP *with a length-increasing p-productive function.* (3) *Every p-productive set for* NP *with a length-increasing p-productive function is p-levelable.* (4) *If A is p-productive for* NP *in* NE, *then A has a length-increasing p-productive function.*

REMARK 3.24. Statements 1, 2, and 3 in Lemma 3.23 are also true if we replace "length-increasing" for "p-honest."

From Lemma 3.23, we know that if A is p-productive for NP in NE, then A is p-levelable. So from this result and Theorem 3.21 we can easily derive a proof for the following theorem, which holds for NTIME($T(n)$) as well.

THEOREM 3.25. *If there exists a p-productive set for* NP *in* NE, *then every $\leq_m^p$-complete set for* NE *is p-levelable.*

THEOREM 3.26. *If there exists a p-productive set for NP in NTIME($T(n)$), then every $\leq_m^p$-complete set for* NTIME($T(n)$) *is p-levelable.*

4. Levelable NP sets. Studying NP sets is more difficult, at least in part because it is not known whether or not there is a universal predicate within NP for all polynomial-time computable functions. Joseph and Young [5] defined a class of creative sets within NP by considering fixed polynomial degrees. We study p-approximations to NP sets using the concept of k-creativity. Let $\mathcal{NM}^k = \{M_i : (\forall x \in L(M_i))[M_i \text{ accepts } x \text{ within } |i||x|^k + |i| \text{ steps}]\}$. Let $\text{NP}^k = \text{NTIME}(n^k)$. Then clearly, $\text{NP}^k = \{L_i : M_i \in \mathcal{NM}^k\}$.

A set A is *k-creative* [14] if $A \in \text{NP}$ and there is a total polynomial-time computable function f, which is called k-productive function, such that $(\forall i)[M_i \in \mathcal{NM}^k \Rightarrow [L_i \subseteq \overline{A} \Rightarrow f(i) \in \overline{A} - L_i]]$.

A set A is *k-completely creative* [5] if $A \in \text{NP}$ and there is a total polynomial-time computable function f, which is called k-completely productive function, such that $(\forall i)$ $[M_i \in \mathcal{NM}^k \Rightarrow [f(i) \in A \Leftrightarrow f(i) \in L_i]]$.

The complement of a k-creative set is called *k-productive*. All k-completely creative sets were shown to be NP-complete in Joseph and Young [5]. However, we do not know whether or not all k-creative sets are NP-complete. From the definition, it is easy to see that k-completely creative sets are k-creative. But it is not known whether every k-creative set is l-completely creative for some $l > 0$ [14], [16]. A set is honest k-creative (k-completely creative) if it is k-creative (k-completely creative) and has a p-honest p-productive (p-completely productive) function.

It was shown in [11] that honest k-completely creative sets are p-levelable. On the other hand, it was shown in Homer [3] that the complement of every honest k-completely creative set contains an infinite subset in NP^l for some $l > 0$. Homer asked whether the complement of every honest k-completely creative set contains infinite subset in P. We settle this problem by showing a stronger result.

The following lemma, due to Book, Greibach, and Wegbreit [1] (see also [13]), will be used to prove Theorem 4.2. The lemma indicates that for nondeterministic time complexity we can get by with TMs having a fixed number of tapes. No similar result is known for deterministic time complexity [13]. We can prove Theorem 4.2 without using this lemma by considering one-tape Turing machines in the proof. For details see Wang [15].

LEMMA 4.1 (see Book, Greibach, and Wegbreit [1]). *For each* TM M *there is a* 2-*tape* TM M' *and a constant* c *such that* $L(M') = L(M)$ *and* $T_{M'}(x) \leq cT_M(x)$ *for every* $x \in L(M)$, *where* c *is proportional to the number of tapes in* M.

THEOREM 4.2. *The complement of every honest k-creative set is p-levelable.*

Proof. We first show that the complement of every honest k-creative set contains an infinite subset in P. Let A be honest k-creative. Then there is a total, p-honest polynomial-time computable function h such that for all i, if $M_i \in \mathcal{NM}^k$, then $L_i \subseteq \overline{A} \Rightarrow h(i) \in \overline{A} - L_i$. So for all i, if $M_i \in \mathcal{NM}^k$, then we have

$$L_i \subseteq \overline{A} \Rightarrow h(i) \in \overline{A} - L_i. \tag{3}$$

Suppose M_h is a DTM computing h within $O(n^l)$ steps on input of length n. By Lemma 4.1, define a two-tape TM M such that on input (x, y), M simulates M_x on y. If M_x accepts y within $|x||y|^k + |x|$ steps or $M_h(x) = y$, then M accepts; otherwise, M rejects. Since the number of tapes of M_x is less than $|x|$ and the program x is read for at most $|x||y|^k + |x|$ times during the simulation, by Lemma 4.1, we can have $T_M(x, y) \leq O(|x|^2(|x||y|^k + |x|) + |x|^l)$. By the s-m-n theorem, there is a total polynomial-time computable function g such that $M_{g(x)}$ is a two-tape TM, $M_{g(x)}(y) = M(x, y)$, and $T_{M_{g(x)}}(y) \leq O(T_M(x, y))$.

So for all x, $T_{M_{g(x)}}(y) \leq O(|x|^3|y|^k + |x|^{\max\{3,l\}})$. By padding g we can get a new two-tape program f such that f is total, polynomial-time computable, length-increasing, $(\forall x)(\forall y)[M_{f(x)}(y) = M_{g(x)}(y)]$, and $T_{M_{f(x)}}(y) = T_{M_{g(x)}}(y) \leq |f(x)||y|^k + |f(x)|$. That is, $M_{f(x)} \in \mathcal{NM}^k$ for all x. By the construction, if $M_x \in \mathcal{NM}^k$, then $M_{f(x)}$ is a two-tape TM, $M_{f(x)} \in \mathcal{NM}^k$, and

$$L_{f(x)} = L_{g(x)} = L_x \cup \{h(x)\}. \tag{4}$$

From (3) and (4), similar to the proof of Lemma 3.6, the reader can easily construct an infinite subset of $\overline{A}$ in P.

Similar to the proofs of Lemmas 3.7 and 3.8, it is easy to complete the proof of Theorem 4.2. □

COROLLARY 4.3. *Every honest k-completely creative set is bi-p-levelable.*

5. Final remarks and open problems. We have shown that p-productivity and k-creativity are powerful concepts for studying polynomial-time approximations to intractable sets. Some open problems have been mentioned as they appear. The most interesting ones among them are listed as follows.

1. Do there exist p-productive sets for NP in NE?
2. Is every k-creative set honest k-creative?
3. Do there exist k-productive sets in NP?

It was shown in [11] that sets not in P that are either paddable or self-reducible are p-levelable. So it would be interesting to investigate what relationships hold among p-paddability, self-reducibility, and p-creativity or p-productivity.

Acknowledgment. I am grateful to Steven Homer for reading the earlier drafts of this paper and for his support when I was in Boston University. I would like to thank Tim Long and Paul Young for their encouragement on this work. My special thanks are due to Alan Selman and the complexity theory group at Northeastern University. I thank Eric Allender for his suggestion and the anonymous referees for their helpful comments.

REFERENCES

[1] R. BOOK, S. GREIBACH, AND B. WEGBREIT, *Time- and tape-bounded Turing acceptors and AFLs*, J. Comput. System Sci., 6 (1970), pp. 606–621.
[2] L. BERMAN, *On the structure of complete sets: almost everywhere complexity and infinitely often speedup*, Proc. 17th Annual IEEE Symposium on Foundations of Computer Science, 1976, pp. 76–83.
[3] S. HOMER, *On simple and creative sets in* NP, Theoret. Comput. Sci., 47 (1986), pp. 169–180.
[4] J. HOPCROFT AND J. ULLMAN, *Introduction to Automata Theory, Languages, and Computation,* Addison-Wesley, Reading, MA, 1979.
[5] D. JOSEPH AND P. YOUNG, *Some remarks on witness functions for nonpolynomial and noncomplete sets in* NP, Theoret. Comput. Sci., 39 (1985), pp. 225–237.
[6] K. KO, *Nonlevelable sets and immune sets in the accepting density hierarchy in* NP, Math. Systems Theory, 18 (1985), pp. 189–205.
[7] K. KO AND D. MOORE, *Completeness, approximation and density,* SIAM J. Comput., 10 (1981), pp. 787–796.
[8] M. MACHTEY AND P. YOUNG, *An Introduction to the General Theory of Algorithms,* North-Holland, Amsterdam, 1978.
[9] A. MEYER AND M. PATERSON, *With what frequency are apparently intractable problems difficult?* Tech. Report TM-126, Laboratory for Computer Science, MIT, Cambridge, MA, 1979.
[10] P. ORPONEN, *A classification of complexity core lattices*, Theoret. Comput. Sci., 47 (1986), pp. 121–130.
[11] P. ORPONEN, D. RUSSO, AND U. SCHÖNING, *Optimal approximations and polynomially levelable sets*, SIAM J. Comput., 15 (1986), pp. 399–408.

[12] K. REGAN, *Minimum-complexity pairing functions,* Tech. Report 88-92, Cornell Mathematical Sciences Institute, Ithaca, NY; to appear in J. Comput. System Sci.

[13] J. SEIFERAS, M. FISCHER, AND A. MEYER, *Separating nondeterministic time complexity classes*, J. Assoc. Comput. Mach., 25 (1978), pp. 146–167.

[14] J. WANG, *P-creative sets vs. p-completely creative sets,* Proc. of the 4th Annual IEEE Conference on Structure in Complexity Theory, June 1989, pp. 24–33.

[15] ———, *P-productivity and polynomial time approximations,* Proc. of the 5th Annual IEEE Conference on Structure in Complexity Theory, July 1990, pp. 254–265.

[16] ———, *On P-creative sets and p-completely creative sets,* Theoret. Comput. Sci., 85 (1991), pp. 1–31.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1112–1129, December 1992
© 1992 Society for Industrial and Applied Mathematics
007

OPTIMAL REDUCTION OF TWO-TERMINAL DIRECTED ACYCLIC GRAPHS*

WOLFGANG W. BEIN†, JERZY KAMBUROWSKI‡, AND MATTHIAS F. M. STALLMANN§

Abstract. Algorithms for series-parallel graphs can be extended to arbitrary two-terminal dags if *node reductions* are used along with series and parallel reductions. A node reduction contracts a vertex with unit in-degree (out-degree) into its sole incoming (outgoing) neighbor. This paper gives an $O(n^{2.5})$ algorithm for minimizing node reductions, based on vertex cover in a transitive auxiliary graph. Applications include the analysis of PERT networks, dynamic programming approaches to network problems, and network reliability. For NP-hard problems one can obtain algorithms that are exponential only in the minimum number of node reductions rather than the number of vertices. This gives improvements if the underlying graph is nearly series-parallel.

Key words. algorithms, complexity, NP-completeness, directed acyclic graph, series-parallel graph, transitive graph, triconnected components, reliability, dynamic programming, PERT network

AMS(MOS) subject classifications. 05C20, 05C75, 05C85, 68M15, 68Q20, 68Q25, 68R10, 90B25, 90C35, 90C39

1. Introduction. Duffin [9] proved that a two-terminal directed acyclic graph (st-dag) is series-parallel if and only if it does not contain a subgraph homeomorphic from the graph pictured in Fig. 1, the *interdictive graph*. Series-parallel st-dags can be efficiently parsed and transformed into a decomposition tree (see [38]). Many graph and network problems that are either intractable or have complicated solutions in the general case are easy in the special case of series-parallel networks. Bein, Brucker, and Tamir [4], for example, show that the minimum cost flow problem is solved by the greedy algorithm if and only if the graph is series-parallel. Other examples include scheduling and sequencing problems [1], [2], [26], [27], location problems [17], as well as many combinatorial problems [21], [35], [34]. All these approaches rely on the decomposition tree (see [6] for a general formulation of this idea) or on Duffin's characterization.

This paper introduces definitions of st-dag *complexity*, measures that describe how nearly series-parallel an st-dag is. One can then, for hard problems, extend the work described in the previous paragraph to obtain algorithms that are exponential only in the complexity of the underlying st-dag, rather than its size.

Our primary measure is motivated by Duffin's characterization. We eliminate all embedded interdictive graphs by successive node reductions, and the number of node reductions determines the complexity. Our main result is that there exists a polynomial-time algorithm to minimize node reductions, obtained by showing that the complexity of an st-dag is equal to the size of a minimum vertex cover in a transitive auxiliary graph.

A second measure of complexity, derived from the first and motivated by network dynamic programming, is based on *factoring*, a generalization of the decomposition tree to arbitrary st-dags.

In §2 we introduce both complexity measures, motivate them with examples, and show that factoring complexity is bounded by reduction complexity. Section 3 gives several equivalent definitions of the auxiliary graph. Section 4 contains details relating a

*Received by the editors July 18, 1988; accepted for publication (in revised form) September 16, 1991.
†American Airlines Decision Technologies, MD 4462 HDQ, P. O. Box 619616, Dallas/Fort Worth Airport, Texas 75261-9616.
‡Department of Information Systems and Operations Management, University of Toledo, Toledo, Ohio 43606-3390.
§Department of Computer Science, North Carolina State University, Raleigh, North Carolina 27695-8206.
This author's research was partially supported by Office of Naval Research contract N00014-88-K-0555, which is gratefully acknowledged.

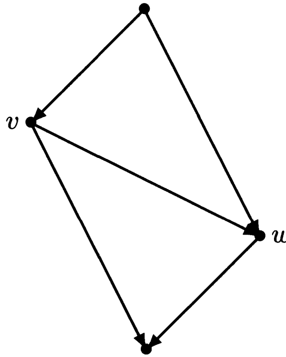


FIG. 1. *The interdictive graph* (IG).

vertex cover in the auxiliary graph to reduction complexity. In §5 we present an $O(n^{2.5})$ algorithm to compute an optimal node reduction sequence and, therefore, the reduction complexity. Finally, §6 presents some open problems.

We conclude this section with definitions and notation that will be used throughout the paper. We already introduced the term st-dag to refer to a two-terminal directed acyclic graph. An st-dag always has a unique source s and a unique sink t. This implies that an st-dag is connected, i.e., there is a path from s to any vertex and from any vertex to t. In our notation, the st-dag G is regarded as a set of edges (multiple edges between the same two vertices are permitted), while $V(G)$ denotes the set of vertices in G.

When we say an st-dag is *series-parallel* we mean that it is two-terminal edge series-parallel (see [38] for a description of the relationship between edge and vertex series-parallel). More specifically, an st-dag is series-parallel if it can be obtained iteratively in the following way:

- A single edge is two-terminal series-parallel (with the tail being the source and the head being the sink);
- If G_1 and G_2 are two-terminal series-parallel, so is the graph obtained by identifying the sources and sinks, respectively (parallel composition);
- If G_1 and G_2 are two-terminal series-parallel, so is the graph obtained by identifying the sink of G_1 with the source of G_2 (series composition).

The in-degree of vertex v in G is denoted by $in(v, G)$, while $out(v, G)$ denotes the out-degree. The notation $v < w$ means that there is a path from v to w in G; we use $v \leq w$ if equality is a possibility. $P(v, w)$ refers to a particular path from v to w, specifically to the set of vertices on that path.

A vertex v *dominates* another vertex w if every path from s to w includes v. Conversely, if every path from v to t includes w, then w *reverse-dominates* v. These definitions are symmetric in the sense that v dominates w in G if and only if v reverse-dominates w in G^R (G with directions of all edges reversed, and s and t interchanged). Vertex v *properly dominates* w if v dominates w and $v \neq w$. *Properly reverse-dominates* is defined similarly.

A dag H is *homeomorphic from* another dag H' if H can be obtained from H' by repeatedly inserting vertices of in-degree and out-degree one in the middle of edges (edges of H' are transformed into disjoint paths of H). We say that a dag G has an IG (*interdictive graph*) at v, w if G has a subgraph homeomorphic from the graph in Fig. 1 with v and w in the positions shown. It is well known that an st-dag G is series-parallel if and only if there are no IGs in G [9].

2. Definitions of dag complexity. Our primary definition of complexity refers to a sequence of reductions of an st-dag. A *parallel reduction* at v, w replaces two or more edges $e_1, \cdots, e_k$ joining v to w by a single edge $g = (v, w)$. A *series reduction* at v is possible when $e = (u, v)$ is the unique edge into v, and $f = (v, w)$ is the unique edge out of v: then e and f are replaced by $g = (u, w)$. A *node reduction* at v can occur when v has in-degree or out-degree 1 (a node reduction is a generalization of a series reduction). Suppose v has in-degree 1, and let $e = (u, v)$ be the edge into v. Let $f_1 = (v, w_1), \cdots, f_k = (v, w_k)$ be the edges out of v. Replace $\{e, f_1, \cdots, f_k\}$ by $\{g_1, \cdots, g_k\}$, where $g_i = (u, w_i)$. The case where v has out-degree 1 is symmetric: here $e = (v, w)$, $f_i = (u_i, v)$, and $g_i = (u_i, w)$.

For convenience, let $G \circ v$ denote the result of a node reduction at v, and let $[G]$ denote the graph that results when all possible series and parallel reductions have been applied to G (this is well defined because series and parallel reductions obey the Church–Rosser property: the order in which reductions are applied does not affect the final outcome [37]). An st-dag G is said to be *irreducible* if $[G] = G$.

DEFINITION 2.1. Let $\mu(G)$, the *reduction complexity* of G, be the minimum number of node reductions sufficient (along with series and parallel reductions) to reduce G to a single edge. More precisely, $\mu(G)$ is the smallest c for which there exists a sequence $v_1, \cdots, v_c$ such that $[\cdots [[[G] \circ v_1] \circ v_2] \cdots \circ v_c]$ is a single edge. Such a sequence is called a *reduction sequence*.

Finding $\mu(G)$ and the corresponding optimal reduction sequence is important in the solution of several problems defined on deterministic and stochastic networks: estimating completion time in PERT networks, travel time in transportation networks, and network reliability. More details and references relating to these applications are given in [10]. We illustrate the basic ideas by presenting details of the simplest application, two-terminal reliability.

Let G be an st-dag in which each edge e is assigned a failure probability $p(e)$. Then $R(G)$, the *two-terminal reliability* of G, is the probability that there exists at least one source-sink path with no failed edges in G. Determining $R(G)$ is #P-complete even when G is planar with maximum degree 3, and all failure probabilities are the same [28]. If G is series-parallel, $R(G)$ can be computed in linear time using series and parallel reductions [3]. For each reduction, the failure probability of the new edge can be computed so that the network has the same reliability before and after the reduction. When a series reduction replaces edges e and f with g,

$$p(g) = 1 - (1 - p(e))(1 - p(f)).$$

When a parallel reduction replaces $e_1, \cdots, e_k$ with g,

$$p(g) = p(e_1) \cdots p(e_k).$$

Eventually the st-dag is reduced to a single edge e^* with $R(G) = 1 - p(e^*)$.

We extend the result of [3] to obtain an $O(m2^c)$ time algorithm for computing the reliability of complexity-c st-dags. Consider a node reduction replacing $\{e, f_1, \cdots, f_k\}$

by $\{g_1, \cdots, g_k\}$. Let G be the st-dag before the node reduction; let G' be the st-dag after the node reduction with $p(g_i) = p(f_i), i = 1, \cdots, k$; let G'' be $G - \{e, f_1, \cdots, f_k\}$ (the reduced node and all its incident edges are deleted). That is, G' is derived from G under the condition that e does not fail, while G'' is what G turns into when e fails. To make G'' an st-dag, we also remove from it all vertices and edges that are not on any path from s to t. Then we apply the recurrence

$$R(G) = (1 - p(e))R(G') + p(e)R(G'').$$

A recursive algorithm using this formulation and the rules for series and parallel reductions has a worst case running time of $O(m2^c)$ arithmetic operations when there are c node reductions. In practice the time is likely to be much better, since G'' often has lower complexity than G', and many series and parallel reductions reduce the size of the st-dag between successive node reductions. (See also [32] and [39] for a description of graph reduction techniques used for undirected reliability problems.)

We now introduce a second complexity measure, based on a generalization of the decomposition tree to arbitrary dags. If G is an st-dag, it is possible to define a (not necessarily unique) algebraic expression α for the set of all source-sink paths in G. The expression α, also called a *factoring* of G, consists of edges and the operators + (disjoint union) and $\cdot$ (concatenation, also denoted by juxtaposition when no ambiguity arises). If G is series-parallel, + corresponds to parallel composition and $\cdot$ to series composition, and it is possible to obtain a unique factoring (up to applications of associative laws — see [38]).

The definition of a factoring can be made more formal, as follows. A *path expression* α is an algebraic expression for a set of paths between two specific vertices of an st-dag. Let $\mathcal{P}(\alpha)$ denote the set of paths represented by α, and let $s(\alpha)$ be the start vertex for all paths in $\mathcal{P}(\alpha)$ while $t(\alpha)$ is the terminal vertex. Any valid path expression is obtained by recursively applying the following rules.

- A single edge e from v to w is a path expression with $\mathcal{P}(e) = \{e\}$, $s(e) = v$, and $t(e) = w$.
- If α_1 and α_2 are path expressions with $s(\alpha_1) = s(\alpha_2)$, $t(\alpha_1) = t(\alpha_2)$, and $\mathcal{P}(\alpha_1) \cap \mathcal{P}(\alpha_2) = \emptyset$, then $\alpha_1 + \alpha_2$ is a path expression with $s(\alpha_1 + \alpha_2) = s(\alpha_1)$, $t(\alpha_1 + \alpha_2) = t(\alpha_1)$, and $\mathcal{P}(\alpha_1 + \alpha_2) = \mathcal{P}(\alpha_1) \cup \mathcal{P}(\alpha_2)$.
- If α_1 and α_2 are path expressions with $t(\alpha_1) = s(\alpha_2)$, then $\alpha_1 \cdot \alpha_2$ is a path expression with $s(\alpha_1 \cdot \alpha_2) = s(\alpha_1)$, $t(\alpha_1 \cdot \alpha_2) = t(\alpha_2)$, and $\mathcal{P}(\alpha_1 \cdot \alpha_2) = \{P_1P_2 | P_1 \in \mathcal{P}(\alpha_1), P_2 \in \mathcal{P}(\alpha_2)\}$.

Both $+$ and $\cdot$ are associative, and $\cdot$ has precedence over $+$. A *factoring* of an st-dag G is a path expression α for which $\mathcal{P}(\alpha)$ is the set of all source-sink paths in G (also $s(\alpha) = s$ and $t(\alpha) = t$).

Unlike the series-parallel case (where a factoring is a linear representation of the series-parallel decomposition tree), we allow factorings to have duplicate occurrences of subexpressions. For example, two possible factorings of the st-dag in Fig. 2 are $(ad + b)g + ((ad + b)f + ((ad + b)e + c)h)i$, which repeats the expression $ad + b$, and $(ad + b)(ehi + fi + g) + chi$, which repeats the expressions hi and i.

DEFINITION 2.2. We say that α *duplicates* an expression α' if α' appears more than once, say k times, in α and any larger expression α'' containing α' appears less than k times (i.e., α' is a maximal duplicated expression). The *cost* of a factoring α is the number of distinct expressions duplicated by α. The minimum cost of any factoring of an st-dag G is called the *factoring complexity* of G, denoted by $\psi(G)$.

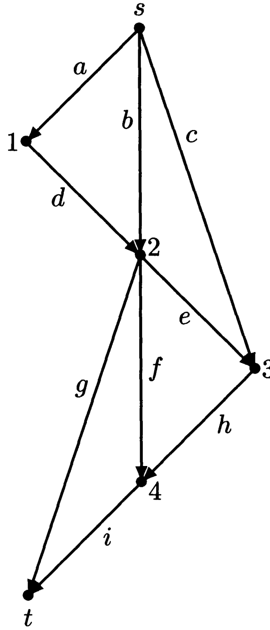


FIG. 2. *An st-dag.*

In figuring the cost, it is not relevant whether there are only two or arbitrarily many occurrences of a specific expression. For example, the cost of the first factoring is only 1, even though $ad + b$ occurs 3 times. The cost of the second factoring is 2.

Factoring complexity is central to the efficient solution by dynamic programming of many network optimization problems. We discuss one of these in detail to motivate Definition 2.

Suppose the edges of the st-dag $G = (V, E)$ represent activities in a project, and the duration of an activity e is given by the function $\mathcal{G}(e, x)$, where x is the number of units of some resource R that are devoted to the activity e. Assume $\mathcal{G}$ is defined only when x is an integer in the range $0 \cdots U - 1$, so for each fixed e the values of $\mathcal{G}$ can be represented by a vector of length U. The problem is to find an allocation of R among the activities so that the overall project duration is minimized. Robinson [31] observed that if G is series-parallel, the minimum duration over all possible allocations can be found by dynamic programming (the optimum allocation can be derived easily from the computation). More precisely, we can extend $\mathcal{G}$ to path expressions as follows, where $\mathcal{G}(\alpha, x)$ is the maximum total duration of any path in $\mathcal{P}(\alpha)$, given that x units of R are allocated optimally among the activities in α.

$$\mathcal{G}(\alpha_1 + \alpha_2, x) = \min_{0 \le r \le x} \{\max(\mathcal{G}(\alpha_1, r), \mathcal{G}(\alpha_2, x - r))\}, \tag{1}$$

(2) $$\mathcal{G}(\alpha_1 \cdot \alpha_2, x) = \min_{0 \le r \le x} \{\mathcal{G}(\alpha_1, r) + \mathcal{G}(\alpha_2, x - r)\}.$$

These formulations are correct only if $\mathcal{G}$ can be optimized independently on α_1 and α_2, that is, if no activities occur in both α_1 and α_2. In this case, we say that $\mathcal{G}$ is *separable* with respect to α_1 and α_2.

Robinson also pointed out that if the number of units of R to be allocated to certain judiciously chosen activities is fixed in advance, $\mathcal{G}$ may be rendered separable even if G is not series-parallel. Of course, in computing the minimum duration, all possible ways of fixing the allocation of these activities must be considered, so that the algorithm based on (1) and (2) must be executed $O(U^c)$ times, where c is the number of activities whose allocation is fixed in advance.

Consider, for example, the st-dag G in Fig. 3 and one of its factorings $\alpha = ((ac+b)e+ad)g+(ac+b)f$. To render $\mathcal{G}(\alpha, x)$ completely separable, the allocation to activities a, b, and c must be fixed. It is possible to show that 3 is the minimum number of fixed activities required to render any factoring of this example separable. Suppose, however, that we consider fixing the allocation to whole subexpressions rather than just single activities. Only two expressions $ac + b$ and a are duplicated by α, so we can rewrite α as follows:

$$\begin{aligned} \alpha_1 &= a, \\ \alpha_2 &= \alpha_1 c + b, \\ \alpha &= (\alpha_2 e + \alpha_1 d)g + \alpha_2 f. \end{aligned}$$

Now we can render $\mathcal{G}(\alpha, x)$ separable by fixing the allocation to α_1 and α_2. The number of ways to do this is $O(U^2)$ rather than $O(U^3)$. Let r_1 be the number of units of R assigned to α_1, and let r_2 be the number of units assigned to α_2 (not including the r_1 units assigned to α_1). Then the optimal allocation for G is computed as follows (e_1 and e_2 denote two surrogate activities whose durations are fixed at $\mathcal{G}(\alpha_1, r_1)$ and $\mathcal{G}(\alpha_2, r_2)$, respectively):

$$\begin{aligned} \mathcal{G}(\alpha, x) &= \min_{0 \le r_1 + r_2 \le x} \{\mathcal{G}((e_2 e + e_1 d)g + e_2 f, x - r_1 - r_2)\}, \\ \mathcal{G}((e_2 e + e_1 d)g + e_2 f, x) &= \min_{0 \le r \le x} \{\max(\mathcal{G}((e_2 e + e_1 d)g, r), \mathcal{G}(e_2 f, x - r))\}, \\ \mathcal{G}((e_2 e + e_1 d)g, x) &= \min_{0 \le r \le x} \{\mathcal{G}(e_2 e + e_1 d, r) + \mathcal{G}(g, x - r)\}, \\ \mathcal{G}(e_2 e + e_1 d, x) &= \min_{0 \le r \le x} \{\max(\mathcal{G}(e_2 e, r), \mathcal{G}(e_1 d, x - r))\}, \\ \mathcal{G}(e_2 e, x) &= \mathcal{G}(\alpha_2, r_2) + \mathcal{G}(e, x), \\ \mathcal{G}(\alpha_2, r_2) &= \min_{0 \le r \le r_2} \{\max(\mathcal{G}(\alpha_1 = a, r_1) + \mathcal{G}(c, r), \mathcal{G}(b, r_2 - r))\}, \\ \mathcal{G}(e_1 d, x) &= \mathcal{G}(a, r_1) + \mathcal{G}(d, x), \\ \mathcal{G}(e_2 f, x) &= \mathcal{G}(\alpha_2, r_2) + \mathcal{G}(f, x). \end{aligned}$$

Intuitively a factoring of minimum cost also minimizes the number of possible fixed allocations that need to be considered when computing project duration (we give no formal proof here).

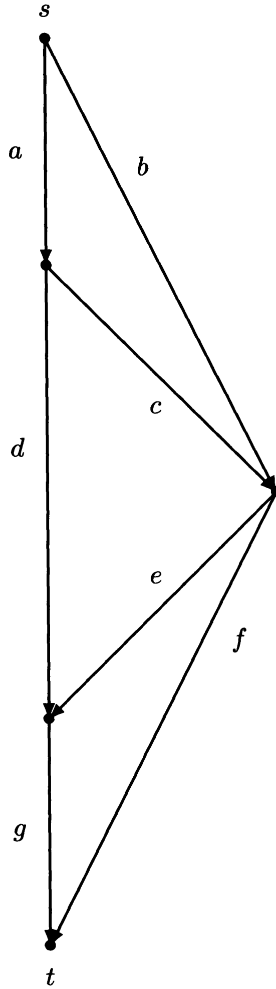


FIG. 3. *Another st-dag.*

The two complexities, factoring complexity and reduction complexity, are related (the idea described in the remainder of this paragraph comes from [7]). From a sequence of series, parallel, and node reductions reducing G to a single edge, we can obtain a fac-

toring as follows. Initially we label every edge e of G with the label $S(e) = e$. Ultimately the single edge e^* to which G is reduced has a label $S(e^*)$ giving a factoring of G. The new label for the edge g resulting from a series reduction of e and f is $S(g) = S(e) \cdot S(f)$. For a parallel reduction, $S(g) = S(e_1) + \cdots + S(e_k)$. For each new edge g_i resulting from a node reduction (see earlier definition), let $S(g_i) = S(e) \cdot S(f_i)$ when $in(v, G) = 1$ or $S(g_i) = S(f_i) \cdot S(e)$ when $out(v, G) = 1$. This has the effect of creating as many duplicate copies of $S(e)$ as there are edges leading out of (into) v. Figure 4 shows the reduction process used to derive the first factoring of the st-dag in Fig. 2. Because the source and sink are never reduced and at least two intermediate vertices are required for an interdictive graph, we have the following result.

THEOREM 2.3. *If G is a* st-dag *with n nodes, then* $\psi(G) \leq \mu(G) \leq n-3$. *We conjecture, but have not been able to prove, that* $\psi(G) = \mu(G)$.

3. The auxiliary graph. The key to computing reduction complexity is an auxiliary graph that identifies the IGs, both explicit and hidden, that need to be eliminated. Informally an edge between v and w in the auxiliary graph implies that any reduction sequence must either reduce v or w (or both). We proceed by defining this auxiliary graph.

DEFINITION 3.1. The *complexity graph*, $C(G)$, of an st-dag G, is defined by $(v, w) \in C(G)$ if and only if there exists a path $P(v, w)$ in G such that for every $u \in P(v, w)$; u neither properly dominates w nor properly reverse-dominates v. Equivalently, $u \in P(v, w) - \{w\}$ does not dominate w and $u \in P(v, w) - \{v\}$ does not reverse-dominate v.

It is easy to see that this definition is equivalent to the following.

DEFINITION 3.2. $(v, w) \in C(G)$ if and only if $v < w$, no $x \geq v$ properly dominates w, and no $x \leq w$ properly reverse-dominates v.

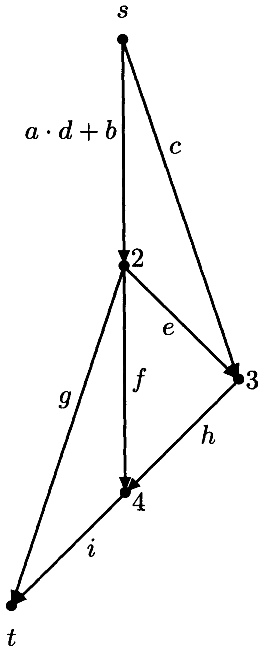
We assume $V(C(G))$ is the set of all vertices that are endpoints of edges in $C(G)$, i.e., $C(G)$ has no isolated vertices. Note that the definitions imply $out(v, G) > 1$ and $in(w, G) > 1$. An edge of $C(G)$ is identified only by its endpoints, i.e., there are no parallel edges in $C(G)$. Neither s nor t appears as a vertex of $C(G)$, and hence $C(G)$ has at most $|V(G)| - 2$ vertices. While the complexity graph is still directed and acyclic, it is not an st-dag. It is easy to show that $C([G]) = C(G)$, from which it follows that $C(G)$ is empty if and only if G is series-parallel. Figure 5 shows the complexity graph for the dag in Fig. 2.

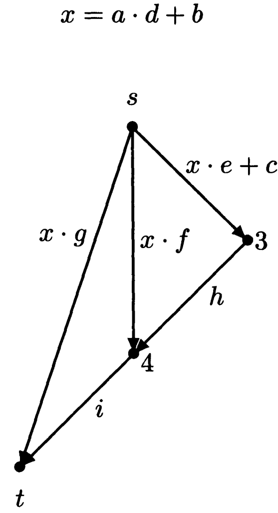
LEMMA 3.3. *$C(G)$ is a transitive graph.*

Proof. Suppose (u, v) and (v, w) are edges of $C(G)$. Let $P(u, v)$ and $P(v, w)$ be the paths satisfying Definition 3.1. We claim that $P(u, w) = P(u, v)P(v, w)$ also satisfies the definition. Note that any vertex of $P(u, v)$ that dominates w must also dominate v, and any vertex of $P(v, w)$ that reverse-dominates u must also reverse-dominate v. □

The two definitions of $C(G)$ are equivalent to a characterization in terms of homeomorphic subgraphs: $(v, w) \in C(G)$ if G has a subgraph homeomorphic from one of the four dags depicted in Fig. 6 with v and w and the indicated position (we also include cases where the edge e in the series IG or the compound IG is contracted, i.e., x_1 coincides with x_2). Observe that there are two types of hidden IGs in the four subgraphs. With the series IG there are IGs at v, x_1 and x_2, w, but a node reduction at x_1 or x_2 does not eliminate either; thus, a reduction at v or w is forced. With the parallel IG a node reduction at v' or w' is possible, but either reduction creates a new IG at v, w. The compound IG combines both phenomena.

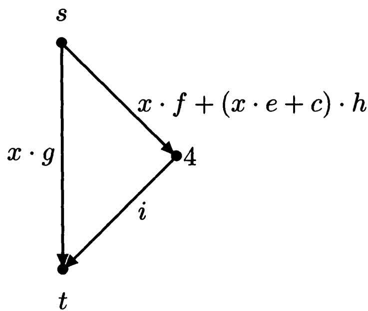
The ability to reduce all cases requiring node reduction to one of these four is what originally led us to believe in the existence of a polynomial-time algorithm. The discovery of the parallel IG and the compound IG forced us to reject the simple conjecture that

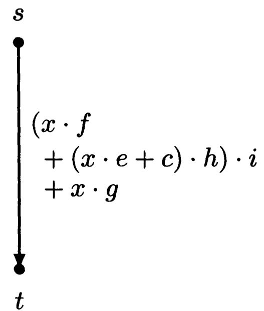




series reduction at 1
parallel reduction at $s, 2$

node reduction at 2
parallel reduction at $s, 3$





series reduction at 3
parallel reduction at $s, 4$

series reduction at 4
parallel reduction at s, t

FIG. 4. *The relationship between reduction and factoring.*

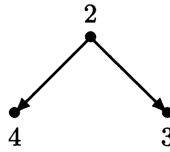
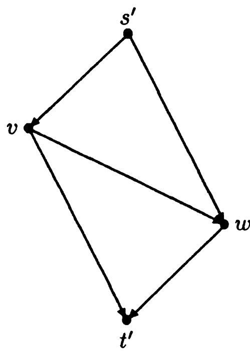
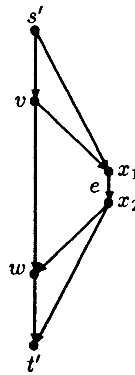


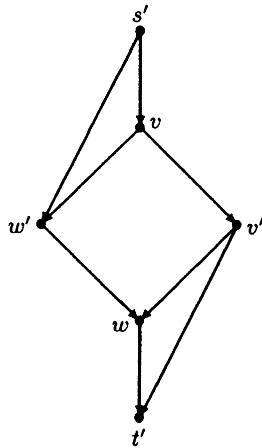
FIG. 5. *The complexity graph of the* st-dag *in Fig.* 2.



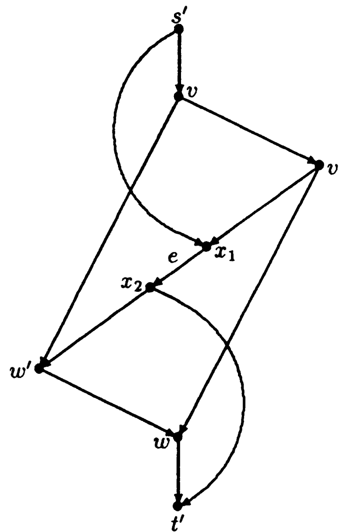
The IG.



The series IG.



The parallel IG.



The compound IG.

FIG. 6. *Four cases with pairwise constraints on* v, w.

$C(G)$ is derived by taking the transitive closure of a dag in which (v, w) is an edge if and only if there is an IG at v, w. Using the following lemma, which gives yet another definition of $C(G)$, it is easy to verify that $(v, w) \in C(G)$ implies existence of one of the four homeomorphic subdags.

LEMMA 3.4. *The edge (v, w) is in $C(G)$ if and only if there exist paths $P_1(v, w)$, $P_2(v, w)$, $P(v, t)$, and $P(s, w)$ such that $P_1(v, w) \cap P(v, t) = \{v\}$ and $P_2(v, w) \cap P(s, w) = \{w\}$. Furthermore, either $P_1(v, w) = P_2(v, w)$ or $P_1(v, w) \cap P_2(v, w) = \{v, w\}$, and the edges common to $P(s, w)$ and $P(v, t)$, if any, form a single path.*

See [5] for a detailed proof. The four dags correspond to four combinations obtained by (a) letting $P_1(v, w)$ and $P_2(v, w)$ either be disjoint or identical, and (b) letting $P(s, w)$ and $P(v, t)$ either be disjoint or intersecting.

4. The auxiliary graph and node reduction. We now prove that the minimum number of node reductions required to reduce an st-dag G to a single edge is equal to the cardinality of a minimum vertex cover in $C(G)$. The easy part of this is showing that the vertex cover size is a lower bound on $\mu(G)$.

LEMMA 4.1. *If c is the cardinality of a minimum vertex cover in $C(G)$, then $c \leq \mu(G)$.*

Proof. It suffices to show that, whenever $(v, w) \in C(G)$, an arbitrary reduction sequence for G must include either v or w. Suppose $(v, w) \in C(G)$ and consider a reduction sequence $v_1, \cdots, v_k$ that includes neither v nor w. We argue by induction that $(v, w) \in C(G_i)$ for $i = 1, \cdots, k$, where $G_i = [\cdots [[[G] \circ v_1] \circ v_2] \cdots \circ v_i]$. The conclusion is that $(v, w) \in C(G_k)$, i.e., G_k is not a single edge and the sequence does not reduce G.

Obviously $(v, w) \in C(G_0)$ since $G_0 = G$. Assume $(v, w) \in C(G_{i-1})$, and observe that a node reduction does not introduce any new dominance relations among the vertices that remain. Using the fact that $v_i \notin \{v, w\}$ and Definition 3.2, we see that $(v, w) \in C(G_{i-1} \circ v_i)$. Since $C(G) = C([G])$, we can conclude $(v, w) \in C(G_i)$. This completes the induction argument. □

The remainder of the section shows how a node reduction sequence for G may be computed from a vertex cover of $C(G)$. If V' is a vertex cover of $C(G)$, we have to ensure that at every stage of reducing G at least one vertex of V' is eligible for node reduction (has in-degree or out-degree equal to 1). The arguments that follow show, essentially, that an eligible vertex is always present in some triconnected component of G.

An st-dag $G(v, w) \subseteq G$ is an *autonomous subdag* in G if it has source v and sink w and satisfies one additional property: For every path P from s to t in G, the set of edges $P \cap G(v, w)$ is either empty or forms a path from v to w. Note that v may be s and w may be t or both, but we disallow the two trivial cases where $G(v, w)$ is a single edge or all of G. If $G(v, w)$ is an autonomous subdag in G, we call v and w *split vertices* of G. We use the notation $G/G(v, w)$ to denote the st-dag G, but with $G(v, w)$ replaced by the edge (v, w).

Some observations about autonomous subdags are in order. First note that v dominates every vertex in $G(v, w)$ and w reverse-dominates every vertex in $G(v, w)$. If $G(v, w)$ has exactly two edges, then $G/G(v, w)$ corresponds to the result of either a series reduction or a parallel reduction of G. It is not hard to show that $C(G) = C(G/G(v, w)) \cup C(G(v, w))$, where the two terms of the union are vertex and edge disjoint.

If we assume the edge (s, t) is always present in G (this edge has no effect on our complexity measures), then a recursive decomposition of G into autonomous subdags (decomposing G into $G/G(v, w)$ and $G(v, w)$ at each level) corresponds to a decomposition of the underlying undirected graph into triconnected components (using the definition of MacLane [24]). A decomposition of G into autonomous subdags could, therefore, be found in linear time using a minor modification of the Hopcroft–Tarjan

algorithm [19]. Such a decomposition is not required for our algorithms.

Finally, in most of our applications it suffices to solve the problem separately for each autonomous subdag, replacing the subdag by a single edge that carries the information about the subdag. When used in practice our definition of st-dag complexity should be modified to take this observation into account. Specifically, we should take $\mu(G)$ to be the maximum reduction complexity of any subdag encountered in the decomposition, and apply Theorem 4.6 (below) to each component of $C(G)$.

The following lemma allows us to conclude (within each autonomous subdag) that sources and sinks of $C(G)$ correspond to vertices eligible for node reduction in G.

LEMMA 4.2. *If $v \in V(G) - \{s,t\}$ is not a split vertex of G and v has in-degree (out-degree) greater than 1 in G, then v has in-degree (out-degree) greater than 0 in $C(G)$.*

Proof. Suppose that $in(v,G) > 1$ (the argument for out-degree is symmetric). Then there exists $s' < v$ (in G) with at least two disjoint paths between s' and v and every path from s to v includes s' (note: s' may be the same as s). Let $U = \{u \in V(G) \mid u < v \text{ and } u > s'\}$. There must exist $u \in U$ and a path $P(u,t)$ such that $P(u,t) \cap U = \{u\}$, and $v \notin P(u,t)$. Otherwise G has an autonomous subdag with source s' and sink v, and v is a split vertex. The path $P(u,t)$ and the two disjoint paths from s' to v ensure that no x with $u \leq x \leq v$ either properly dominates v or properly reverse-dominates u. Thus, by Definition 3.2, $(u,v) \in C(G)$ and v has in-degree greater than 0 in $C(G)$. □

An edge (v,w) of a dag G is said to be *contractable* if v has out-degree 1 and w has in-degree 1. If (v,w) is contractable, let $G/(v,w)$ denote the dag obtained by contracting the edge (v,w), identifying vertices v and w. The following arguments imply that a node reduction along a contractable edge is superfluous.

LEMMA 4.3. *If (v,w) is a contractable edge of G, then $C(G/(v,w)) = C(G)/(v,w)$, where $C(G)/(v,w)$ denotes $C(G)$ with vertices v and w identified.*

Proof. Note that any paths including v or w must either end at v, begin at w, or include the edge (v,w). Combining v and w into a single vertex therefore has no effect on the dominance relations of Definition 3.2. □

LEMMA 4.4. *If G is irreducible and $V' \neq \emptyset$ is a minimum vertex cover of $C(G)$, then there exists $v \in V'$ such that v is a source or sink of $C(G)$, v is not a split vertex of G, and v is not the endpoint of a contractable edge of G.*

Proof. First note that if G has an autonomous subdag $G(v,w)$, neither of the split vertices v,w appears in $C(G(v,w))$. Note also that $C(G(v,w))$ must have at least one edge. Otherwise $G(v,w)$ is series-parallel and G is not irreducible. Since V' must include at least one vertex of $C(G(v,w))$, we can reduce our search to a component $C(H)$ of $C(G)$ such that there are no autonomous subdags in H and $C(H)$ is nonempty (i.e., H is a minimal autonomous subdag of G).

To find a vertex of $V' \cap V(H)$ that is a source or sink of $C(H)$ (and thus of $C(G)$) and not the endpoint of a contractable edge, repeatedly do contractions along every contractable edge of H and call the resulting graph $\hat{H}$. It follows from Lemma 4.3 that $C(\hat{H}) \neq \emptyset$ — if $(v,w) \in C(H)$, then (v,w) cannot be a contractable edge of H, nor can it become one as the result of other contractions. Since $C(\hat{H})$ is transitive (Lemma 3.3) and nonempty, at least one source and one sink of $C(\hat{H})$ must be connected by an edge $(\hat{s},\hat{t})$. We claim that $(\hat{s},\hat{t})$ is also an edge of $C(H)$. Note that when two vertices v and w are combined in $C(\hat{H})$ as the result of a contraction (see Lemma 4.3), the new combined vertex cannot be a source or a sink of $C(\hat{H})$. Otherwise, by Lemma 4.2, one of v,w had in-degree 1 and out-degree 1, contradicting the fact that G is irreducible. Therefore, both $\hat{s}$ and $\hat{t}$ are vertices of H and not new vertices resulting from contraction. Lemma 4.3 implies that the edge $(\hat{s},\hat{t})$ could not have been added as the result of a contrac-

tion, and hence must be an edge of $C(H)$. This means that at least one of $\hat{s},\hat{t}$ must be in V'. □

The following lemma ensures that properly chosen node reductions will decrease reduction complexity. Let $C(G) \setminus v$ be $C(G)$ with v and all incident edges deleted.

LEMMA 4.5. *If G is irreducible, v is eligible for node reduction in G (either $in(v, G)$ or $out(v, G)$ is 1), and v is not the endpoint of a contractable edge, then $C(G \circ v) \subseteq C(G) \setminus v$.*

Proof. Suppose $in(v, G) = 1$ (the argument for out-degree is symmetric). It is easy to see that a node reduction at v, i. e., the contraction of an edge (u, v), does not remove dominance relations among vertices other than u and v. Thus it suffices to show that (u, w) is in $C(G)$ whenever $(u, w) \in C(G \circ v)$. Since we know (u, v) is not a contractable edge and $in(v, G) = 1$, we know that $out(u, G) > 1$; hence v does not reverse-dominate u in G. Thus the insertion of v into a path $P(u, w)$ does not alter the fact that it satisfies Definition 3.1. □

THEOREM 4.6. *If c is the cardinality of a minimum vertex cover in $C(G)$, then $\mu(G) = c$.*

Proof. Because of Lemma 4.1 it suffices to show that $\mu(G) \leq c$, which we do by induction on c. Assume, without loss of generality, that G is irreducible—recall that $C([G]) = C(G)$ and $\mu([G]) = \mu(G)$.

The basis case $c = 0$ is immediate from earlier remarks. If $c > 0$, let V' be the vertex cover of cardinality c in $C(G)$. By Lemma 4.4 there exists a $v \in V'$ such that v is a source or sink of $C(G)$, and v is neither a split vertex nor the endpoint of a contractable edge. From Lemma 4.2 we know that $in(v, G) = 1$ or $out(v, G) = 1$, i.e., a node reduction at v is possible. Lemma 4.5 allows us to conclude that $C([G \circ v])$ has a vertex cover of cardinality no greater than $c - 1$. Hence the desired result follows by induction. □

5. Polynomial time algorithms. In this section we establish the time bound for computing $\mu(G)$, the minimum number of node reductions required to reduce an st-dag G. We show that computing $C(G)$ is equivalent to computing the transitive closure of G. Because $C(G)$ is a transitive dag, we can compute a minimum vertex cover in $C(G)$ by reducing the problem to finding a maximum matching in a bipartite graph (the complement of a minimum vertex cover is a maximum independent set, which in a transitive dag corresponds to a Dilworth chain decomposition — see [14] for details). The overall time bound for computing $\mu(G)$ is, therefore, $O(n^{2.5})$.

Computing the actual reduction sequence for an st-dag G is straightforward.

> compute $C(G)$ as described below
>
> compute V', a minimum vertex cover in $C(G)$
>
> **while** $V' \neq \emptyset$ **do**
>
> > perform all series/parallel reductions possible in G and let $G := [G]$
> >
> > find $v \in V'$ such that $in(v, G)$ or $out(v, G)$ is 1
> >
> > > and v is not the endpoint of a contractable edge
> >
> > $G := G o v$
> >
> > $V' := V' - \{v\}$
>
> **enddo**

Based on Definition 3 we observe that $C(G)$ is the intersection of two dags, $C_1(G)$

and $C_2(G)$, defined as follows:

$$C_1(G) = \{(v, w) \mid v < w \text{ and no } x \geq v \text{ properly dominates } w\},$$

$$C_2(G) = \{(v, w) \mid v < w \text{ and no } x \leq w \text{ properly reverse-dominates } v\}.$$

So the problem of computing $C(G)$ reduces to computing $C_1(G)$ and $C_2(G)$. Since $C_1(G)$ and $C_2(G)$ are symmetric, we discuss only the computation of $C_1(G)$.

Suppose $T(G)$, the tree of dominators in G, has been computed. This can be done in time $O(m + n)$, where m is the number of edges in G [16]. Recall that v is the parent of w in $T(G)$ if and only if v properly dominates w and there does not exist v' such that v properly dominates v' and v' properly dominates w. Note: s is the root of $T(G)$. Let $C_1(v) = \{ w \mid (v, w) \in C_1(G) \}$, and note that $w \in C_1(v)$ if and only if $w > v$ and no ancestor x of w in $T(G)$ has $x \geq v$. It follows that for any vertex v the following two steps compute $C_1(v)$: (1) mark all $w \geq v$ (all other vertices are unmarked), and (2) do a preorder traversal of $T(G)$ to find all marked nodes that have no marked ancestors (v constitutes a special case and is not included in $C_1(v)$). Steps (1) and (2) can each be done in time $O(n)$ if the transitive closure of G has been computed (the transitive closure is used in step (1)). We, therefore, have the following.

THEOREM 5.1. *$C(G)$ can be computed in time $O(n^2 + M(n))$, where $M(n)$ is the time required for computing the transitive closure of a graph of n vertices (a recent upper bound for $M(n)$ is $O(n^{2.37})$ [8]).*

Since minimum vertex cover in a transitive graph is equivalent to maximum matching in a bipartite graph [14], we have the following (see [18] for the bipartite matching algorithm).

THEOREM 5.2. *The minimum number of node reductions required to reduce an* st-dag *G to a single edge, $\mu(G)$, can be computed in time $O(n^{2.5})$.*

This time bound is not likely to improve significantly unless the time bounds for bipartite matching and transitive closure are improved. A simple reduction shows that (a) computing $\mu(G)$ is at least as hard as finding the size of a minimum vertex cover in the transitive closure of a dag, and (b) computing $C(G)$ is at least as hard as computing transitive closure. Let G be an arbitrary dag, and define an st-dag G' by adding a new source s and a new sink t to G with edges (s, v) and (v, t) for every vertex v in G. It is easy to see that $(v, w) \in C(G')$ if and only if $(v, w) \in G^*$, where G^* is the transitive closure of G. Thus computing the auxiliary graph is as hard as computing transitive closure. By previous arguments $\mu(G') = c$, where c is the cardinality of a minimum vertex cover in $G^* = C(G')$. Thus computing reduction complexity is as hard as finding the size of a minimum vertex cover in the transitive closure.

6. Open problems. As we observed in §2, efficient algorithms based on series-parallel reduction extend to graphs that are nearly series parallel. Problems that are NP-hard in general can be solved in polynomial time for st-dags of fixed (reduction) complexity c, where st-dags of complexity 0 are series-parallel. This paper gives a polynomial-time recognition algorithm for complexity-c st-dags (note that the algorithm is polynomial even when c is not fixed). In §2 a series-parallel two-terminal reliability algorithm was extended to run in polynomial time for st-dags of fixed complexity c. Similar techniques can be applied to some NP-hard fault diagnosis problems (see [36] for the series-parallel algorithms), and probably many other st-dag problems.

However, these techniques are not suited to most NP-hard scheduling problems. The dag in a scheduling problem defines a precedence relation and is in general not an st-dag. At stake here is the relationship between activity-on-arc and activity-on-node

(partial order or precedence) representations of activity networks (see, e. g., [11]). The techniques of §2, when used to solve PERT network problems, assume the input to be an activity-on-arc network. Unfortunately, the activity-on-arc representation is not unique: a given project may have many different activity-on-arc representations while having only one activity-on-node representation. Past research has focused on translating from activity-on-node to activity-on-arc representation so as to minimize the number of "dummy activities," a problem that is NP-hard (see [22]). We have demonstrated that sometimes minimizing complexity is a more suitable objective than minimizing dummy activities. In this connection Michael [25] has recently shown that the translation to a minimum complexity activity-on-arc network can be done in polynomial time if the node set of the output is fixed (minimizing dummy activities is still NP-hard under the same assumption).

Optimal translation to activity-on-arc representation is not an issue for *vertex series-parallel digraphs*, which translate directly to series-parallel st-dags (see [38]), or for line digraphs (see [15] for a characterization), which have a unique activity-on-arc representation with no dummy activities. While there does not appear to be any natural analogue of node reduction to complement the series and parallel reductions for vertex series-parallel dags, the decomposition into autonomous subdags used in the proofs of §4 corresponds to the *modular decomposition* of transitive dags defined by Spinrad [33] (modular decomposition is defined for undirected graphs, but extends naturally to dags defining partial orders).

The idea of using node reduction to augment series and parallel reduction appears to be most promising for problems traditionally formulated on undirected graphs. For problems such as independent set and dominating set, where the object is to find an optimal subset of the vertices, we can simply try out two possibilities for each vertex removed by node reduction (either the vertex is in the optimal solution or it's not), using the techniques of [21], [35] to deal with series or parallel reductions. This gives $O(m2^c)$ time algorithms for maximum independent set, minimum dominating set, and other undirected graph problems when these problems are formulated on st-dags. Similarly, we can obtain an $O(mK^c)$ time algorithm for K-coloring (see [34]).

How does the notion of complexity extend to undirected graphs? With the help of an st-*numbering*, a numbering of the vertices in which vertex 1 is adjacent to vertex n and each other vertex has at least one lower numbered and one higher numbered neighbor [12], [23], each biconnected component of an undirected graph can be treated as an st-dag. Each undirected edge is oriented from a lower to a higher st-number. The complexity of an undirected graph can thus be defined as the minimum complexity given by any st-numbering. Use the maximum complexity of any biconnected component if there is more than one (actually, for most of these problems it suffices to use the maximum complexity of each *triconnected* component). Since there are, in the worst case, up to $n!$ possible st-numberings, the recognition problem for complexity-c undirected graphs may be difficult (unless $c = 0$, when it is equivalent to series-parallel recognition). Figure 7 shows an undirected graph whose complexity is either 1 or 2, depending on which st-numbering is used (note that both numberings have the same source and sink and that the graph is triconnected).

It is interesting to note that for fixed c, the recognition problem for complexity-c undirected graphs can be shown to be in P via a nonconstructive argument, using the work of Robertson and Seymour [29], [30] (see also [13] and [20]). The key observation is that the complexity of an undirected graph never increases when an edge is deleted or contracted. Two open problems are: (a) find a polynomial-time algorithm for the recog-

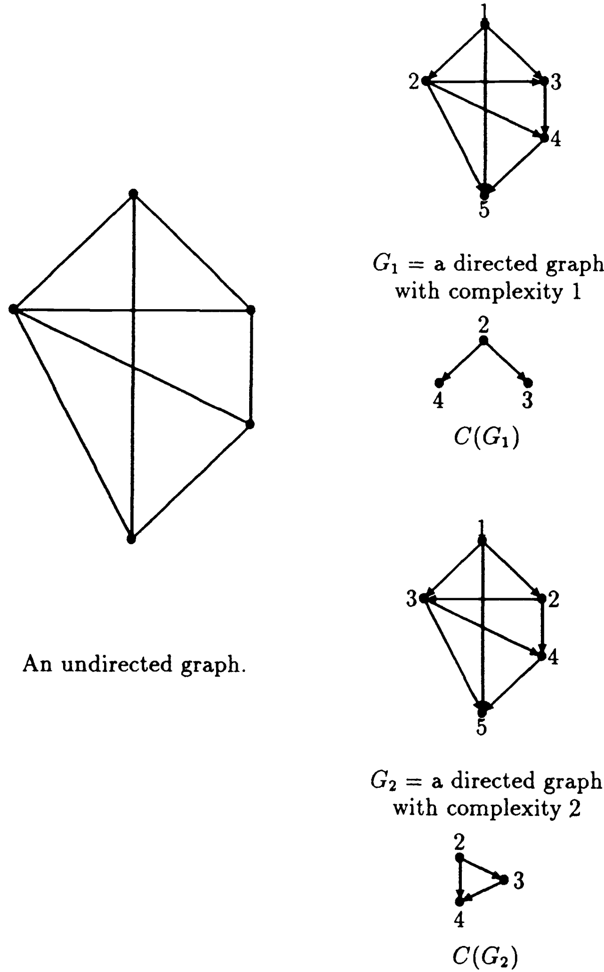


FIG. 7. *Extending complexity to undirected graphs.*

nition of complexity c undirected graphs when c is a small fixed value, and (b) determine whether recognizing complexity c undirected graphs is NP-complete when c is part of the input.

There are also open problems related to the use of node reduction in various applications. We have shown, for example, that $\mu(G)$ is an upper bound for $\psi(G)$, the factoring complexity defined in §2. We believe it is a lower bound as well, i.e., the two complexities are equal, but the proof is difficult because arbitrary factorings are difficult to characterize. It is not hard to show that $(v, w) \in C(G)$ implies that every factoring of G either duplicates an expression ending at v or one beginning at w. Factorings that are not derived from a node reduction sequence appear to have higher cost than those that are, but we have been unable to prove this.

Acknowledgment. We thank Salah Elmaghraby for introducing us to this fascinating problem, Bob Tarjan for many helpful comments on an earlier draft of this paper, and the referees for several careful readings and many insightful comments. We especially appreciate the suggestions by one of the referees for Definition 3.2 and the simpler

algorithm for computing $C(G)$ in §5.

REFERENCES

[1] H. ABDEL-WAHAB AND T. KAMEDA, *Scheduling to minimize maximum cumulative cost subject to series-parallel precedence constraints*, Oper. Res., 26 (1978), pp. 141–158.

[2] ———, *On strictly optimal schedules for the cumulative cost-optimal scheduling problem*, Computing, 24 (1980), pp. 61–86.

[3] A. AGRAWAL AND A. SATYANARAYANA, *An $O(|E|)$ time algorithm for computing the reliability of a class of directed networks*, Oper. Res., 32 (1984), pp. 493–515.

[4] W. BEIN, P. BRUCKER, AND A. TAMIR, *Minimum cost flow algorithms for series-parallel networks*, Discrete Appl. Math., 10 (1985), pp. 117–124.

[5] W. BEIN, J. KAMBUROWSKI, AND M. STALLMANN, *Alternate characterizations of the complexity graph*, Tech. Report 91-21, Department of Computer Science, North Carolina State University, Raleigh, NC, 1991.

[6] M. BERN, E. LAWLER, AND A. WONG, *Linear-time computation of optimal subgraphs of decomposable graphs*, J. Algorithms, 8 (1987), pp. 216–235.

[7] A. COLBY AND S. ELMAGHRABY, *On the complete reduction of directed acyclic graphs*, OR Report 197, Operations Research Program, North Carolina State University, Raleigh, NC, May 1984.

[8] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progressions*, in Proceedings 19th Annual ACM Symposium on Theory of Computing, New York, NY, 1987, pp. 1–6.

[9] R. DUFFIN, *Topology of series-parallel networks*, J. Math. Anal. Appl., 10 (1965), pp. 303–318.

[10] S. ELMAGHRABY, J. KAMBUROWSKI, AND M. STALLMANN, *On the reduction of acyclic digraphs and its applications*, Tech. Report 233, Operations Research Program, North Carolina State University, Raleigh, NC, August, 1989.

[11] S. E. ELMAGHRABY, *Activity Networks: Project Planning and Control by Network Models*, John Wiley & Sons, New York, 1978.

[12] S. EVEN AND R. TARJAN, *Computing an st-numbering*, Theoret. Comput. Sci., 2 (1976), pp. 339–344.

[13] M. FELLOWS AND M. LANGSTON, *Nonconstructive advances in polynomial-time complexity*, Inform. Process. Lett., 26 (1987), pp. 157–162.

[14] L. FORD, JR. AND D. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[15] F. HARARY AND R. NORMAN, *Some properties of line digraphs*, Rend. Circ. Mat. Palermo, 9 (1960), pp. 149–163.

[16] D. HAREL, *A linear time algorithm for finding dominators in flow graphs and related problems* (*extended abstract*), in Proceedings 17th Annual ACM Symposium on Theory of Computing, Providence, RI, 1985, pp. 185–194.

[17] R. HASSIN AND A. TAMIR, *Efficient algorithms for optimization and selection on series-parallel graphs*, SIAM J. Algebraic Discrete Meth., 7 (1986), pp. 379–389.

[18] J. HOPCROFT AND R. KARP, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.

[19] J. HOPCROFT AND R. TARJAN, *Dividing a graph into triconnected components*, SIAM J. Comput., 2 (1973), pp. 135–158.

[20] D. S. JOHNSON, *The* NP-*completeness column: an ongoing guide*, J. Algorithms, 8 (1987), pp. 285–303.

[21] T. KIKUNO, N. YOSHIDA, AND Y. KOKUDO, *A linear algorithm for the domination number of a series- parallel graph*, Discrete Appl. Math., 5 (1983), pp. 299–311.

[22] M. KRISHNAMOORTHY AND N. DEO, *Complexity of the minimum-dummy-activities problem in a* PERT *network*, Networks, 9 (1979), pp. 189–194.

[23] A. LEMPEL, S. EVEN, AND I. CEDERBAUM, *An algorithm for planarity testing of graphs*, in Theory of Graphs: International Symposium, July 1966, P. Rosenstiehl, ed., Gordon and Breach, New York, 1967, pp. 215–232.

[24] S. MACLANE, *A structural characterization of planar combinatorial graphs*, Duke Math. J., 3 (1937), pp. 460–472.

[25] D. J. MICHAEL, *Optimal Representation of Activity Networks as Directed Acyclic Graphs*, Ph.D. thesis, Operations Research Program, North Carolina State University, Raleigh, NC, 1991.

[26] C. MONMA, *The two-machine maximum flow time problem with series-parallel precedence constraints*, Oper. Res., 27 (1979), pp. 792–798.

[27] C. MONMA AND J. SIDNEY, *Sequencing with series-parallel precedence constraints*, Math. Oper. Res., 4 (1979), pp. 215–224.

[28] J. S. PROVAN, *The complexity of reliability computations in planar and acyclic graphs*, SIAM J. Comput., 15 (1986), pp. 694–702.

[29] N. ROBERTSON AND P. SEYMOUR, *Disjoint paths — a survey*, SIAM J. Algebraic Discrete Meth., 6 (1985), pp. 300–305.

[30] ———, *Graph minors — a survey*, in Surveys in Combinatorics, I. Anderson, ed., Cambridge University Press, London, 1985, pp. 153–171.

[31] D. ROBINSON, *A dynamic programming solution to cost-time trade-off for CPM*, Management Sci., 22 (1975), pp. 158–166.

[32] A. SATYANARAYANA AND R. WOOD, *A linear-time algorithm for computing K-terminal reliability in series-parallel networks*, SIAM J. Comput., 14 (1985), pp. 818–832.

[33] J. SPINRAD, *On comparability and permutation graphs*, SIAM J. Comput., 14 (1985), pp. 658–670.

[34] M. SYSLO, NP-*complete problems on some tree-structured graphs: a review*, in Proceedings WG '83, International Workshop on Graphtheoretic Concepts in Computer Science, M. Nagl and J. Perl, eds., Trauner-Verlag, Linz, Austria, 1983, pp. 342–353.

[35] K. TAKAMIZAWA, T. NISHIZEKI, AND N. SAITO, *Linear-time computatbility of combinatorial problems on series-parallel graphs*, J. Assoc. Comput. Mach., 29 (1982), pp. 623–641.

[36] S. TOIDA, *A graph model for fault diagnosis*, J. Digital Systems, VI (1982), pp. 345–365.

[37] J. VALDES, *Parsing Flowcharts and Series-Parallel Graphs*, Ph.D. thesis, Computer Science Department, Stanford University, Stanford, CA, 1978.

[38] J. VALDES, R. TARJAN, AND E. LAWLER, *The recognition of series parallel digraphs*, SIAM J. Comput., 11 (1982), pp. 298–313.

[39] R. WOOD, *Factoring algorithms for computing K- terminal network reliability*, IEEE Trans. Reliability, R-35 (1986), pp. 269–278.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1130–1152, December 1992

© 1992 Society for Industrial and Applied Mathematics
008

PREEMPTIVE ENSEMBLE MOTION PLANNING ON A TREE*

GREG N. FREDERICKSON† AND D. J. GUAN‡

Abstract. Consider the problem of finding a minimum cost tour to transport a set of objects between the vertices of a tree by a vehicle that travels along the edges of the tree. The vehicle can carry only one object at a time, and it starts and finishes at the same vertex of the tree. It is shown that if objects can be dropped at intermediate vertices along its route and picked up later, then the problem can be solved in polynomial time. Two efficient algorithms are presented for this problem. The first algorithm runs in $O(k + qn)$ time, where n is the number of vertices in the tree, k is the number of objects to be moved, and $q \leq \min\{k, n\}$ is the number of nontrivial connected components in a related directed graph. The second algorithm runs in $O(k + n \log n)$ time.

Key words. motion planning, vehicle routing, graph algorithms, directed minimum spanning tree, preemption

AMS(MOS) subject classification. 68Q25

1. Introduction.

Consider an undirected weighted graph with objects located at various vertices. Associated with each object is a destination vertex, to which that object is to be moved by a vehicle that traverses the edges of the graph. A fundamental problem in motion planning is to determine a tour of minimum cost for the vehicle to transport all objects from their initial positions to their destinations. In the case of general graphs, the problem is NP-hard, even if the vehicle can transport only one object at a time [11]. However, for special applications such as those that arise in robotics, it is reasonable to consider more restricted classes of graphs. In this paper and a companion paper [10] we consider problems where the graphs are trees, with a vehicle that can transport only one object at a time. In this paper we focus on preemptive object movement. By this we mean that objects can be dropped, and picked up and transported at some later time in the transportation. A *drop* is an unloading of an object at a vertex that is not its destination.

We show that the problem can be solved in polynomial time, and present two efficient algorithms for it. Let n be the number of vertices in the tree and k the number of objects to be transported. Our first algorithm runs in $O(k+qn)$ time, where $q \leq \min\{k, n\}$ is the number of nontrivial strongly connected components in a related directed graph. Our second algorithm runs in $O(k + n \log n)$ time, which is better whenever k is $o(qn)$ and q is $\omega(\log n)$. These results contrast with our results in the case in which objects cannot be dropped at the intermediate vertices. In [10] we show that the nonpreemptive version of our problem is NP-hard, and we give polynomial-time approximation algorithms. Note that viewed from the context of discrete job scheduling problems, it is not so surprising that the preemptive version of the problem is polynomial while the nonpre-

*Received by the editors August 9, 1991; accepted for publication (in revised form) November 18, 1991. A preliminary version of this paper appeared as a part of *Ensemble Motion Planning in Trees*, Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, October 1989, pp. 66–71.

†Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907. The research of this author was supported in part by National Science Foundation grants CCR-86202271 and CCR-9001241 and by Office of Naval Research contract N00014-86-K-0689. A portion of this research was done while this author was on sabbatical leave at the International Computer Science Institute, Berkeley, California.

‡Institute of Applied Mathematics, National Sun Yat-Sen University, Kaohsiung, Taiwan 80424, Republic of China. The research of this author was supported in part by National Science Foundation grant CCR-86202271 and by Purdue University under a David Ross Fellowship.

emptive version is NP-hard. See, for example, the work on the problem of scheduling independent tasks on identical processors [20], [17], [13].

Our results compare with those of others as follows. For the case in which the graph is a general graph, Frederickson, Hecht, and Kim have shown that the problem, which they termed the stacker-crane problem, is NP-hard [11]. For the cases in which the graph is either a simple path or a simple cycle and preemption is allowed, Atallah and Kosaraju have shown that the problem can be solved in $O(k+n)$ time [1]. For the cases in which the graph is either a simple path or a simple cycle and preemption is not allowed, Atallah and Kosaraju have presented algorithms that find an optimal solution in $O(k+n\log\beta(n,q))$ and $O(k+n\log n)$ time for path and cycle, respectively [1]. Frederickson has improved the latter time bound to $O(k+n\log\beta(n,q))$ [8].

We note that our problem appears to be a special case of exercise 7 in §5.4.8 of [19], in which a bus moves in a tree-shaped network. However, neither Knuth nor Karp, to whom the problem is attributed, knows of an efficient solution to this problem [16], [18]. In fact, we are able to show that, in the case that the bus has capacity greater than one, the problem is NP-complete even if preemption is allowed [9].

We make a number of observations about the structure of an optimal tour for the problem. In a manner similar to that in [1], we show that an optimal tour of the original problem can be obtained by solving the balanced version of the problem. While the structure of our approach is similar to that in [1], many additional ideas are needed to generate efficient algorithms when the graph is a tree. We introduce the notion of canonical tour, and show that every balanced problem has an optimal tour that is also canonical. This leads to the reduction of our problem, to the problem of finding a directed minimum spanning tree of a certain directed graph. Our second algorithm uses a hierarchical decomposition of the tree to construct a directed graph with fewer arcs, which thus allows the directed minimum spanning tree to be computed faster.

The rest of the paper is organized as follows. In §2, we introduce notation and definitions and discuss the transformation of the problem into a balanced version. In §3, we characterize a canonical solution and present our $O(k+qn)$ time algorithm for the problem. In §4, we present our second algorithm for the problem, which runs in $O(k+n\log n)$ time.

2. Generating a balanced problem. In this section we define the problem, along with the notion of moves, drops, and a transportation. The structure of our approach is similar to that in [1]. Some of the definitions are repeated from [10] for the reader's convenience. In a manner similar to that in [1], we define a balanced version of a problem, and show that an optimal transportation for the original problem can be obtained by solving the balanced version of the problem. Standard terminology of graph theory, such as a directed graph and an Euler tour, is used in our paper, and can be found in Bondy and Murty [3].

An instance P of the motion planning problem on trees consists of a tree $T=(V,E)$, a nonnegative cost $c(e)$ on each edge $e\in E$, a starting vertex $s\in V$, a set of objects O, and an *initial vertex* x_j, and a *destination vertex* y_j for each object $j\in O$. Each object $j\in O$ is initially located at its initial vertex x_j and has to be moved to its destination vertex y_j by a vehicle that traverses the edges of the tree. The vehicle can carry only one object at a time, and the tour must start and finish at vertex s.

We observe that, for every instance P, there is an optimal transportation such that each object visits the vertices on the path from x_j to y_j exactly once and visits no other vertices. If this is not the case, then there is a cycle traversed by some object. We can replace the cycle traversed by that object by a noncarrying move. This modification does

not increase the cost of the transportation, and repeatedly doing this yields a transportation with the desired property.

A *move* is designated by (x, y, c), where x and y are vertices in V and c is an object $j \in O$ or 0. The vehicle moves along the unique path from x to y in the tree T, and carries an object c in the move if $c \neq 0$, and no object otherwise. Thus, a move with $c \neq 0$ is called a *carrying move*, and a move with $c = 0$ is called a *noncarrying move*.

Let Q be a sequence of moves, (v_i, v_{i+1}, c_i), $0 \leq i \leq r$. It is clear that two consecutive moves, (u, v, c) and (v, w, c), can be expressed as (u, w, c). Although in some cases we may want to decompose a move into a sequence of moves, we assume in general that $v_{i+1} \neq v_i$ and $c_{i+1} \neq c_i$ for $0 \leq i \leq r$. For each object $j \in O$, let Q_j be a sequence of moves obtained from Q by deleting every move (v_i, v_{i+1}, c_i) in Q with $c_i \neq j$. An object j is *transported* from x_j to y_j by Q if Q_j is a sequence of moves (u_i, u_{i+1}, j), $0 \leq i \leq t$, with $u_0 = x_j$ and $u_{t+1} = y_j$. If $t > 0$, then the object j is *dropped* by Q at vertices $u_1, u_2, \cdots, u_t$; otherwise, it is not dropped by Q.

A *transportation* Q for P is a sequence of moves (v_i, v_{i+1}, c_i), $0 \leq i \leq r$, such that $v_0 = v_r = s$, $v_{i+1} \neq v_i$, and every object $j \in O$ is transported from its initial position x_j to its destination y_j. The cost $c(Q)$ of a transportation Q is defined to be the sum of the costs of the edges the vehicle traverses. The motion planning problem is to find a transportation with minimum cost for an input instance P.

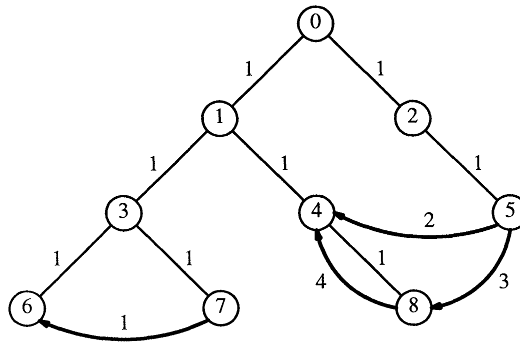


FIG. 1. *An example of the motion planning problem in trees.*

An example of the motion planning problem in trees is given in Fig. 1. There are eight vertices in T and four objects in O. The edges of the tree T are drawn in straight lines. An object j that has to be moved from x_j to y_j is drawn in curved arc from x_j to y_j with label j. The starting vertex is 0. The cost of each edge is 1, as indicated by its label.

We assume that every vertex of degree one or two in T is either s or x_j or y_j for some $j \in O$. A vertex of degree one and the edge incident at it can be deleted from T if it is not s nor x_j or y_j for some $j \in O$. It is easy to see that a vertex of degree two and its adjacent edges can be replaced by a single edge with a cost the sum of the two edges deleted if it is not s nor x_j or y_j for some $j \in O$. Thus, the number of objects k is $\Omega(n)$.

Because every vertex of degree one is either s or x_j or y_j for some $j \in O$, every edge of T must be traversed by a valid transportation at least once. Furthermore, the number of times an edge is traversed in one direction must be equal to the number of times that edge is traversed in the other direction, since the vehicle starts and finishes at s.

Given an optimal transportation Q for a problem P, define a directed graph $D'(Q)$ on the vertex set V such that there is an arc from x to y labeled c if and only if there is a move (x, y, c) in Q. That is, each arc of $D'(Q)$ represents a move of Q. We shall call an arc that represents a carrying move a *carrying arc*, and an arc that represents a

noncarrying move a *noncarrying arc*. It is easy to see that the graph $D'(Q)$ is Eulerian since Q is a transportation that starts and finishes at s. On the other hand, given an instance P, define a directed graph D_0 with vertex set V such that there is an arc from x_j to y_j labeled j if and only if there is an object $j \in O$ initially located at x_j that has to be moved to y_j. If this graph is Eulerian, then any Euler tour starting from s can easily be translated into an optimal transportation for P. Since each arc $\langle x, y\rangle$ in D_0, as well as in $D'(Q)$, represents a move, we assign a cost $d(x, y)$ to it, equal to the sum of the costs of the edges from x to y in T. In a fashion similar to that in [1], the problem is reduced to a special type of graph augmentation problem, that of finding a minimum-cost set of noncarrying moves to add to D_0 to make it Eulerian.

One type of noncarrying moves added are the *balancing moves*. They are added so that every edge is traversed at least once and the number of times an edge is traversed in one direction is equal to the number of times that edge is traversed in the other direction. In the remainder of this section we shall show how to compute a set of balancing moves. Suppose that a set of balancing moves B is given. For each balancing move $(x, y, 0) \in B$, add a *balancing arc* $\langle x, y\rangle$ with label 0 to D_0, and let the resulting graph be D. It is easy to see that the in-degree is equal to the out-degree for every vertex in the graph D, and each connected component of D is thus strongly connected. We shall call the graph D the *balanced graph*. Note that the augmentation by the balancing arcs may not be sufficient to get a transportation.

A strongly connected component of D is called a *trivial component* if it contains only one vertex and this vertex is not s. Otherwise, it is called *nontrivial component*. Note that a nontrivial component that contains x_j or y_j for some $j \in O$ must contain more than one vertex. Since each nontrivial component is Eulerian, no additional noncarrying moves between two vertices in the same nontrivial component are needed. All additional noncarrying moves will be used to connect nontrivial components. We call these noncarrying moves the *linking moves*. We shall show how to find a set of linking moves with minimum cost in the following sections.

There are, in general, many sets of balancing moves of minimum cost that satisfy the above conditions. In [10], we have shown how to construct a set of $O(k + n)$ balancing moves B with minimum cost, and such that the graph D will have a minimum number of nontrivial components. The method is briefly described as follows. First, compute the number of balancing moves required at each edge so that, after these moves are added, every edge is traversed at least once and the number of times an edge is traversed in one direction is equal to the number of times that edge is traversed in the other direction. Second, generate one balancing move on each edge in each direction. Third, generate the remaining balancing moves by merging moves of the from $(u, v_1, o_0), (v_1, v_2, o_1) \cdots (v_t, w, o_t)$ into one move (u, w, o) so that there are at most $O(k + n)$ balancing moves.

Figure 2 shows the balanced problem corresponding to the problem shown in Fig. 1. Although balancing moves are noncarrying moves, we assign for convenience a unique label for each balancing move generated. The moves added are $(3, 1, 5)$, $(1, 3, 6)$, $(3, 7, 7)$, $(6, 3, 8)$, $(4, 1, 9)$, $(1, 0, 10)$, $(0, 2, 11)$, $(2, 5, 12)$, $(4, 0, 13)$, and $(0, 5, 14)$. Note that balancing moves $(3, 1, 5)$ and $(1, 3, 6)$ are added so that the edge $(1, 3)$ is traversed by the vehicle at least once.

In [10], we prove that for every instance P, there is an optimal transportation Q that contains all the moves in the balancing moves B generated by our algorithm. We also show that these balancing moves can be computed in linear time.

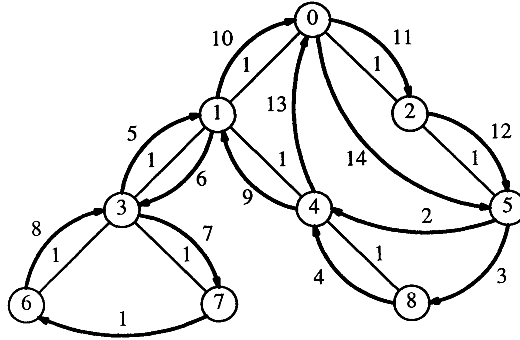


FIG. 2. *A balanced graph (in bold) for the problem in Fig.* 1.

LEMMA 1 [10]. *Given an instance P for the motion planning problem on trees, the balanced graph D for P can be computed in $O(k + n)$ time.*

A motion planning problem is *balanced* if none of the moves in the balanced graph D are balancing moves. Given an instance P, we first construct a set of balancing moves B by the algorithm in [10]. For each balancing move from x to y in B, add an object $o_{x,y}$ to O with initial vertex x and destination vertex y. Let the resulting set of objects be O'. The new instance P' with the objects O' is called the balanced version of the original problem P. We show that adding the balancing moves will not increase the cost of the transportation of the original problem. Part of the proof is similar to that of Lemma 2 in [10], but we must also show that the capability to perform drops does not create a difficulty.

LEMMA 2. *The costs of optimal transportations for P and its balanced version P' are equal.*

Proof. Given a transportation Q' for P', we can obtain a transportation Q for P from Q' by replacing each move (x, y, c) such that $c \notin O$ with a noncarrying move $(x, y, 0)$.

On the other hand, let Q be an optimal transportation for P with a minimum number of drops. First, construct a graph $D'(Q)$ with vertex set V such that there is an arc $\langle x, y \rangle$ labeled c if and only if there is a move (x, y, c) in Q. Since Q is a transportation, graph $D'(Q)$ is Eulerian. Second, replace every arc $\langle x, y \rangle$ in $D'(Q)$ with label 0 by a sequence of arcs $\langle v_i, v_{i+1} \rangle$ labeled 0, $0 \leq i \leq r - 1$, where $x = v_0, v_1, \cdots, v_r = y$ are the list of vertices on the path from x to y in T. The modified $D'(Q)$ will remain Eulerian. Third, for each balancing move from x to y, replace a set of moves $(u_i, u_{i+1}, 0)$, $0 \leq i \leq t$ by a move $(x, y, o_{x,y})$, where $x = u_0, u_1, \cdots, u_{t+1} = y$ are the list of vertices on the path from x to y in T. These moves must exist by the definition of balancing moves. Note that $D'(Q)$ will remain Eulerian under this operation. We claim that any Euler tour of $D'(Q)$ starting from s is a transportation of P'.

The proof of the claim is as follows. It is easy to see that the graph $D'(Q)$ is Eulerian. Therefore, if no objects are dropped by Q, then any Euler tour of $D'(Q)$ starting from s is a transportation of P'. Consider any object j that is dropped at a vertex, say v. Then there must be two arcs with label j, say $\langle u, v \rangle$ and $\langle v, w \rangle$, incident at v. We claim that any directed path from s to v in $D'(Q)$ must contain the arc $\langle u, v \rangle$. Otherwise, we can replace the two arcs $\langle u, v \rangle$ and $\langle v, w \rangle$ by one arc $\langle u, w \rangle$. Since there is a path from s to v that does not contain the arc $\langle u, v \rangle$, the graph $D'(Q)$ would still be connected. But this modification eliminates the drop of object j at vertex v, which is a contradiction of the assumption that Q is a transportation with minimum number of drops. Thus in any

Euler tour of $D'(Q)$, arc $\langle u, v \rangle$ must be traversed before arc $\langle v, w \rangle$. Therefore, any Euler tour of $D'(Q)$ starting from s is a transportation of P'.

Since the split and merge of moves will not change the total cost of the transportation, the cost of Q' is equal to the cost of Q. □

The constructive proof of the above lemma gives a method to translate a transportation for P' into a transportation for P. That is, a transportation for the original problem P can be obtained from the transportation of the balanced problem P' by replacing each move (x, y, c) with $c \notin O$ by $(x, y, 0)$. In the following sections, we discuss how to compute a transportation for the balanced version of the problem.

3. Generating canonical transportations. In this section, we introduce the notion of a canonical transportation and show how it leads to an efficient algorithm for preemptive motion planning problem in a tree. We show how to reduce our problem to the problem of finding a minimum directed spanning tree in a directed graph [2], [4], [5], [6], [12], [22]. This then leads to an $O(k + qn)$ time algorithm.

Given an instance P of the motion planning problem in trees, our algorithm first constructs a balanced graph D as described in §2. Recall that if the balanced graph D is Eulerian and s is not an isolated vertex in D, then any Euler tour of D starting with vertex s is an optimal transportation with no objects dropped. We thus concentrate in this section on how to connect the nontrivial components of D with a minimum cost set of linking moves in the case that D is not Eulerian. Hence we shall assume in this section that a problem P is balanced.

3.1. Bridges and canonical transportations. In this subsection we identify a certain type of transportation, and show that there always exists a transportation of this type that is optimal. We first identify sets of vertices that are related to each strongly connected component of D. We then characterize how any given strongly connected component relates to other strongly connected components. We then define our special type of transportation, which we call canonical. Finally, we show that there is always some canonical transportation that is an optimal transportation.

Let D_i be a nontrivial strongly connected component of D. We first identify sets of vertices that relate to each strongly connected component D_i. Let $j \in O$ be an object with initial vertex x_j and destination vertex y_j. Note that x_j and y_j must be in the same nontrivial component of D. Thus, an object j is an object in D_i if x_j and y_j are both vertices in D_i. Define $IP(D_i)$ to be the set of vertices in D_i, each of which is either the initial vertex for some object in D_i or the start vertex. (We choose the designator *IP* to stand for "initial position.") Also define $V_T(D_i)$ to be the set of vertices each of which will be visited by some object in D_i. Note that every vertex v must be in $V_T(D_i)$ for some component D_i whenever the problem is balanced.

Consider the example shown in Fig. 3. Each straight line represents an edge of T. The cost of $(0, 1)$ is 9 and all the other edges have cost 1. Each curved arc $\langle x_j, y_j \rangle$ with label j, $1 \leq j \leq 7$, represents a carrying arc of D. The starting vertex is 0. Note that the example is a balanced problem. The balanced graph D has four nontrivial components: $D_1 = \{0\}$, $D_2 = \{1, 7\}$, $D_3 = \{5, 8\}$, and $D_4 = \{4, 6, 9\}$. $IP(D_1) = \{0\}$, $IP(D_2) = \{1, 7\}$, $IP(D_3) = \{5, 8\}$, and $IP(D_4) = \{4, 6, 9\}$. Note that $IP(D_i)$ is the same as the vertex sets in the component D_i if the problem is balanced. $V_T(D_1) = \{0\}$, $V_T(D_2) = \{1, 0, 2, 3, 5, 7\}$, $V_T(D_3) = \{5, 8\}$ and $V_T(D_4) = \{4, 2, 3, 6, 9\}$.

We next study how a given strongly connected component D_i relates to other strongly connected components. Let D_i and D_j be two nontrivial components of D. Since the vehicle can drop objects at intermediate vertices, a path from some vertex $u \in V_T(D_i)$

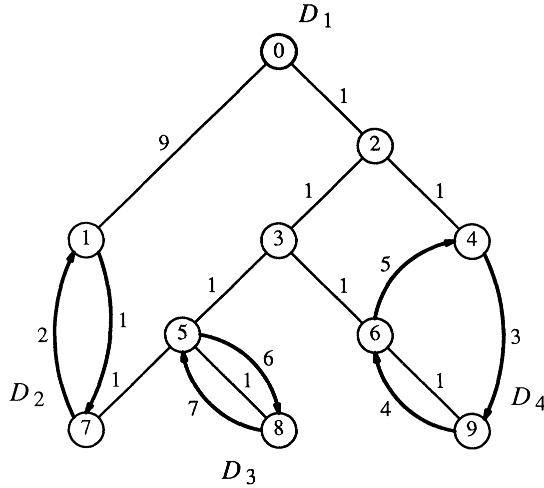


FIG. 3. *A second example of the motion planning problem.*

to some vertex $v \in IP(D_j)$ can be used to link component D_j to component D_i. We call such a path a *bridge* from D_i to D_j. The following example explains how a bridge can be used to connect two components in a transportation.

Assume that the path from u to v is a bridge from D_i to D_j. Let x_o and y_o be the initial and final positions of some object o of D_i such that u is on the path from x_o to y_o. Starting at any vertex in $IP(D_i)$, all objects in D_i and D_j can be transported with at most one drop. The vehicle first transports objects in D_i until it is carrying the object o at vertex u. It then drops object o at u, goes to vertex v, and transports all objects in D_j. After finishing the objects in D_j, the vehicle must be at vertex v. It can then go back to vertex u and pick up object o and finish the rest of the moves of the objects in D_i.

Let each bridge be identified by: (1) the components D_i and D_j that it connects, (2) the origin u and the terminus v of the path, and (3) an object o in D_i such that u is a vertex in the path from x_o to y_o in T. We use $b_{i,j}$ to denote such a bridge from D_i to D_j. Note that, if $u \neq v$, then noncarrying moves $(u, v, 0)$ and $(v, u, 0)$ are the linking moves that are used to connect the components D_i and D_j. If $u = v$, then no linking moves are needed. In either case, if u is not a destination of any object in D_i, then the object o that is associated with the bridge $b_{i,j}$ is dropped at vertex u.

A component D_j is *reachable* from the starting vertex s, with respect to a set of bridges B, if either D_j contains the vertex s or there is a component D_i that is reachable from s with respect to B and there is a bridge $b_{i,j}$ in B from D_i to D_j. If we can find a set B of bridges that make all components reachable from the starting vertex s, then we can compute a transportation by the following procedure.

For each bridge $b_{i,j}$ in B, from D_i to D_j with origin u and terminus v, we add linking arcs $\langle u, v\rangle$ and $\langle v, u\rangle$ to D if the bridge is not a single vertex. Recall that each arc in D represents a move and has a label and a cost. The label of an arc $\langle u, v\rangle$ is an object or 0 and the cost is the distance from u to v in the graph T. The two arcs $\langle u, v\rangle$ and $\langle v, u\rangle$ will have 0 as their labels and distance $d(u, v)$ for their costs. If the vertex u is not a destination of any object in the component D_i, then the object o associated with the bridge will be dropped at vertex u. This is done by splitting the arc $\langle x_o, y_o\rangle$ at vertex u. In general, an object o can be associated with more than one bridge. Thus, the arc $\langle x_o, y_o\rangle$

may be split at more than one vertex. The splits are handled all together, rather than one bridge at a time. We shall show how to do this efficiently after we present the algorithm.

If B is of minimum cost over all sets of bridges that make all components reachable from s, then we call the resulting graph D_B the *augmented balanced graph*. Note that any Euler tour of the augmented balanced graph D_B will traverse the arc $\langle x, u\rangle$ before the arc $\langle u, y\rangle$. This is because u is not a terminus of any arc that can be reached without the bridge $b_{i,j}$. It is easy to see that the augmented balanced graph D_B defines a transportation for P with cost $c(D) + 2c(B)$, where $c(D)$ is the total cost of the arcs in the balanced graph D and $c(B)$ is the total cost of the bridges in B.

Consider the example shown in Fig. 3. Recall that the balanced graph D has four nontrivial components, namely, $D_1 = \{0\}, D_2 = \{1,7\}, D_3 = \{5,8\}$, and $D_4 = \{4,6,9\}$. Let the path from 0 to 4 be the bridge $b_{1,4}$ that connects D_4 from D_1. Let the path from 3 to 5 be the bridge $b_{4,3}$ that connects D_3 from D_4. Let the path from 5 to 7 be the bridge $b_{3,2}$ that connects D_2 from D_3. Note that bridge $b_{3,2}$ is needed, despite the fact that the moves of D_2 pass through vertex 5, since reaching D_2 without initially going through vertex 5 is very expensive. Let 3 be the object that is associated with $b_{4,3}$. All the other bridges start from a destination vertex; thus, the objects associated with them are not used. Since these three bridges make every component reachable from s, we can find a transportation $Q = (0,4,0)(4,3,3)(3,5,0)(5,8,6)(8,5,7)(5,7,0)(7,1,2)(1,7,1)$ $(7,3,0)(3,9,3)(9,6,4)(6,4,5)(4,0,0)$.

Finally, we consider a special type of transportation. We want to show that for any balanced problem P in which objects can be dropped at the intermediate vertices, there is an optimal transportation Q such that each linking move is either the forward or the backward traversal of a bridge. We shall call such a transportation a *canonical transportation*. This reduces our problem to the problem of finding a minimum cost set of bridges that connect the components so that every component can be reached from the starting point s. This is the problem of finding a minimum directed spanning tree in a directed graph, which can be solved efficiently [12], [22].

We first study some properties of an optimal transportation of a balanced motion planning problem on trees that will allow us to prove that there always exists a canonical transportation that is an optimal transportation.

LEMMA 3. *No optimal transportation of a balanced problem can traverse an edge in the same direction more than once without carrying an object.*

Proof. Given a balanced problem P, let Q be an optimal transportation. Without loss of generality, assume that all noncarrying moves of Q are $(x,y,0)$, where (x,y) is an edge of T. Since every vertex in T must be visited by Q, $IP(D_i) = V_T(D_i) = \{v\}$ for each trivial component $D_i = \{v\}$. Consider the set B of the paths that are traversed by the noncarrying moves in Q. Each path in B is a bridge, since every vertex $v \in IP(D_i)$ for some component D_i. Since Q is a transportation, all vertices in T are reachable from s. Thus B contains a set of nonzero-length bridges for the set of all components. Note that no bridges can appear more than once in B, since it could not increase the set of components reachable from s with respect to B. Thus, the lemma follows. □

Given a problem P, let Q be an optimal transportation for P. Let $D'(Q)$ be a directed graph with vertex set V such that there is an arc from x to y labeled c if and only if (x,y,c) is a move in Q. Let $e = (u,v)$ be an edge of T such that e is traversed in Q when the vehicle is not carrying an object. Let $D'_e(Q)$ be a directed graph obtained from $D'(Q)$ by omitting the noncarrying moves on the edge (u,v). That is, replace the noncarrying move $(x,y,0)$ that traverses the edge e in the direction from u to v by two

moves $(x, u, 0)$ and $(v, y, 0)$. Delete any degenerate moves $(u, u, 0)$ and $(v, v, 0)$ that arise whenever $x = u$ and $y = v$, respectively.

LEMMA 4. *Let Q be an optimal transportation for P. Let D'_u and D'_v be the two strongly connected components of $D'_e(Q)$. Then the edge (u, v) is traversed in Q when the vehicle is carrying some object not in the component that contains s.*

Proof. Without loss of generality, assume that s is in D'_u. Since every edge of T is traversed by some object, it is sufficient to show that no objects in D'_u can traverse the edge (u, v). Assume that there were an object o in D'_u that is carried along the edge (u, v). Then v is a bridge from D_u to D_v in $D'_e(Q)$. This would imply that a transportation with smaller cost than Q could be obtained by omitting the noncarrying moves on the edge e. Thus, no objects in D'_u can traverse the edge (u, v). Therefore, the lemma follows. □

LEMMA 5. *Let Q be an optimal transportation of P and $(x, y, 0)$ be the first noncarrying move in Q. Let $x = v_0, v_1, \cdots, v_t = y$ be the sequence of vertices on the path from x to y in T. Let $\bar{P}$ be an instance obtained from P by adding a set of required moves $(x, y, o_{x,y})$ and $(v_i, v_{i-1}, o_{v_i,v_{i-1}})$, $1 \leq i \leq t$ to P. Then optimal transportations for P and $\bar{P}$ have the same cost.*

Proof. Since the optimal transportation Q traverses the path from x to y without carrying an object, it must also traverse every edge on the path from y to x without carrying an object. For each edge (v_i, v_{i-1}) on the (y, x)-path, first find a move $(u, w, 0)$ in Q such that (v_i, v_{i-1}) is in the (u, w)-path in T, and then replace it by $(u, v_i, 0)$, $(v_i, v_{i-1}, 0)$ and $(v_{i-1}, w, 0)$. A transportation Q' of P' can be obtained from Q by replacing each noncarrying move $(v_i, v_{i-1}, 0)$ by $(v_i, v_{i-1}, o_{v_i,v_{i-1}})$, $1 \leq i \leq t$. It is easy to see that Q' and Q have the same cost.

On the other hand, let Q' be an optimal transportation for P'. A transportation Q'' for P can be obtained from Q' by replacing each move $(u, v, o_{u,v})$ such that $o_{u,v}$ is in M', but not in M by $(u, v, 0)$. It is easy to see that Q'' and Q' have the same cost. Therefore, the lemma holds. □

THEOREM 1. *Every balanced problem has an optimal transportation that is canonical.*

Proof. Let P be a counterexample with a minimum number of nontrivial components in the balanced graph. Let Q be an optimal transportation for P and $(u, v, 0)$ be the first noncarrying move in Q. There must be noncarrying moves in Q since an optimal transportation without noncarrying moves is, by definition, canonical. Note that the path from u to v must be a bridge between two nontrivial components of D. Let $u = v_0, v_1, \cdots, v_t = v$ be the sequence of vertices in the path from u to v in T. Add a set of moves $(u, v, o_{u,v})$ and $(v_i, v_{i-1}, o_{v_i,v_{i-1}})$, $1 \leq i \leq t$, to M, and let the resulting instance be $\bar{P}$. By Lemma 5, optimal transportations of P and $\bar{P}$ have the same cost. Since the balanced graph of $\bar{P}$ has fewer nontrivial components than the balanced graph of P, $\bar{P}$ must have a transportation $\bar{Q}$ that is canonical. A canonical transportation Q' for P with the same cost as $\bar{Q}$ can be obtained from $\bar{Q}$ as follows.

Let $D'(\bar{Q})$ be a directed graph with vertex set V such that there is an arc from x to y labeled o if and only if (x, y, o) is a move of $\bar{Q}$. Delete the carrying arcs labeled with objects in $\bar{P}$ but not in P to give $D''(\bar{Q})$. Let $u = u_0, u_1, \cdots, u_r = v$ be the vertices in the path from u to v, upon which there are incident arcs. Let D'_i be the component of $D''(\bar{Q})$ that contains the vertex u_i, $1 \leq i \leq r$. Note that D'_0 is the component that contains s.

For each vertex u_i, $1 \leq i \leq r$, do the following. If there are carrying arcs incident to u_i, that is, u_i an initial position for some objects in some component in D'_i, then add the arcs $\langle u_{i-1}, u_i \rangle$ and the $\langle u_i, u_{i-1} \rangle$ to $D''(\bar{Q})$. Otherwise, there must be two linking arcs

$\langle u_i, v_i' \rangle$ and $\langle v_i', u_i \rangle$ incident on u_i. (Recall that u_i is a vertex on the (u, v)-path at which there are arcs incident on it.) First, find an object o in D_i' that visits the vertex u_i. By Lemma 4, such an object must exist. Second, split the carrying arc $\langle x_o, y_o \rangle$ at u_i. Finally, add the arcs $\langle u_{i-1}, v_i' \rangle$ and $\langle v_i', u_{i-1} \rangle$ to $D''(\bar{Q})$. It is clear that the transformation does not increase the cost of the transportation, and an Euler tour of the resulting $D''(\bar{Q})$ starting with s yields the desired canonical transportation Q' of P. □

Define a *directed bridging graph* A with vertex set the set of nontrivial components of D. For each ordered pair of distinct vertices D_i and D_j, the weight of arc $\langle D_i, D_j \rangle$ is equal to the sum of the costs on the edges of the minimum cost bridge $b_{i,j}$ from D_i to D_j. Let $c(D)$ be the sum of costs of all arcs in D.

THEOREM 2. *A balanced problem P has an optimal transportation with cost $c(D) + 2x$ if and only if the directed bridging graph A has a minimum directed spanning tree of weight x, rooted at the component that contains s.*

Proof. Without loss of generality, let D_1 be the strongly connected component of D that contains s. We first show that a directed spanning tree S, with root D_1 and weight x of A can be translated into a transportation of P with cost $c(D) + 2x$. For each arc $\langle D_i, D_j \rangle$ of S, let $b_{i,j}$ be the corresponding bridge from D_i to D_j, and B be the set of these bridges. It is clear that every nontrivial component of D can be reached from s with respect to the bridges in B. Therefore, P has a transportation of cost $c(D) + 2x$.

We next show that there is an optimal transportation of cost y to the motion planning problem that can be translated into a directed spanning tree with root D_1 and weight $(y - c(D))/2$ of A. Let Q be an canonical and optimal transportation of P. By Theorem 1 there is such a transportation. Construct a directed spanning tree S for A as follows. Examine each move of Q, from the first one to the last. Whenever there is an noncarrying move $(u, v, 0)$ in Q and there is no $\langle D_j, D_i \rangle$ arc in S, we add an arc $\langle D_i, D_j \rangle$ to S, where D_i is the component for the object of the preceding move and D_j is the component for the next move. Note that we also want to add an arc to S for a degenerate bridge. This can be done by examining two consecutive moves of Q. If an object from D_i is in the first move and an object from D_j is in the second move, then there is a degenerate bridge in Q. Add an arc $\langle D_i, D_j \rangle$ to S if the arc $\langle D_j, D_i \rangle$ is not already in S. Since Q must visit all the components of D, S must span all the components. S must be a tree, otherwise we could delete some bridges that correspond to an arc of a cycle in S. The resulting set of arcs would still make all components reachable from s and we could generate a transportation which has less cost then Q, which is a contradiction. By Lemma 3, an optimal transportation must traverse an edge zero times, or two times, once in each direction, without carrying any object. Therefore, the weight of S is equal to $(y - c(D))/2$. □

With the above theorem, our problem is reduced to finding a minimum directed spanning tree with root D_1 that contains s in the bridging graph A. In the next subsection, we shall present an efficient algorithm for the problem.

Consider once again the example in Fig. 3. In Fig. 4 we give the corresponding bridging graph A. There is a node for each of the strongly connected components, D_1, D_2, D_3, and D_4. Consider the nodes D_1 and D_4. The minimum-cost bridge from D_1 to D_4 is the path in T from vertex 0 to vertex 4. Thus the cost of arc $\langle D_1, D_4 \rangle$ is 2. Note that the cost function is not symmetric, since the minimum-cost bridge from D_4 to D_1 is the path in T from vertex 2 to vertex 0, of cost 1. Also note that some arcs have cost 0, as does $\langle D_2, D_3 \rangle$, since vertex 5 is both in the set $V_T(D_2)$ and in the set $IP(D_3)$. The other arcs correspond to minimum-cost bridges that are easily identified. The arcs in a directed minimum spanning tree are shown in bold, and have a total cost of 4.

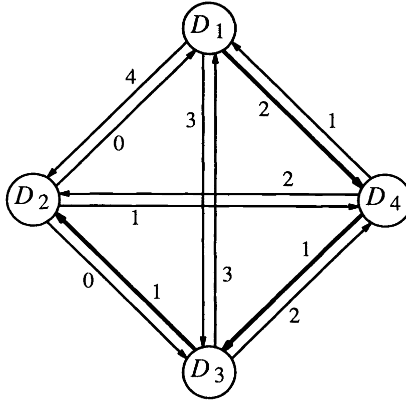


FIG. 4. *The directed bridging graph for Fig.* 3, *with a directed minimum spanning tree in bold.*

3.2. An efficient algorithm. In this section we present the algorithm *with-drops* and show that it can be implemented to run in $O(k+qn)$ time. We first present the algorithm in a high-level description, and then discuss how to implement it efficiently. In particular, we discuss carefully how to compute the directed bridging graph A efficiently. We also compute efficiently how moves should be interrupted in the case that several drops must be made on the same move. Finally, we analyze the time required by the algorithm.

We first present our algorithm. Recall that the directed bridging graph was defined between Theorems 1 and 2 in the preceding subsection.

ALGORITHM *with-drops*
INPUT: an instance P of motion planning problem on trees.
OUTPUT: an optimal transportation Q for P.
METHOD:

1. Find the balanced graph D for the motion planning problem P.
2. Find the directed bridging graph A for D, rooted at the node representing the component that contains the start vertex.
3. Find a minimum directed spanning tree B of the graph A.
4. Find the augmented balanced graph D_B for D with bridges in B.
5. Output a transportation Q by finding an Euler tour of D_B starting from s.

We first show how to construct the bridging graph efficiently. The algorithm processes one nontrivial component of D at a time. Let D_i be a nontrivial component. With each bridge $b_{i,j}$ we associate the following information: the components D_i and D_j, the origin u and the terminus v of the path, and an object o in D_i such that o must visit the vertex u in the transportation. For each vertex u in $V_T(D_i)$, we use $a_i(u)$ to denote such an object. We show how to compute the vertex set $V_T(D_i)$. Note that the value of $a_i(v)$ for every vertex v in $V_T(D_i)$ can also be computed as we compute the vertex set $V_T(D_i)$.

Let s be the root of T. For each object o in D_i, let t be the nearest common ancestor of x_o and y_o. If t is not x_o or y_o, then replace the carrying arc $\langle x_o, y_o\rangle$ by two arcs $\langle x_o, t\rangle$ and $\langle t, y_o\rangle$, both labeled with the object o. Reorient each arc so that every arc is directed from a child toward an ancestor. For each vertex v in T, let *into*(v) be the list of arcs with terminus v, and let *outof*(v) be a list of arcs with origin v. Each entry in *into*(v) stores the address of the corresponding entry in *outof*(u) that represents the arc $\langle u, v\rangle$. Each entry in *outof*(u) stores the name of the arc that the entry represents. The lists *into*(v)

and *outof*(v) for all vertices in T can be constructed in $O(k+n)$ time. Given these lists, we then call the recursive procedure *search* with parameter s.

For each vertex v, the procedure *search* determines if v is in $V_T(D_i)$ and computes the value $a_i(v)$ by maintaining a list of arcs $L(v)$ such that v is a vertex of the path from x to y for every arc $\langle x, y\rangle$ in $L(v)$. A vertex v is in $V_T(D_i)$ if and only if $L(v)$ is not empty. Given $L(v)$, the value $a_i(v)$ can also be computed in constant time, e.g., the label of the first arc in the list $L(v)$.

The procedure *search*(v) does the following. If v is a leaf, then let $L(v) =$ *outof*(v). Otherwise, if v is not a leaf, do the following. First, let $L(v)$ be empty. Second, for each child w of v, call *search*(w) and merge $L(w)$ to $L(v)$. Third, if $L(v)$ is not empty, then add v to $V_T(D_i)$, and let $a_i(v)$ be the original name of the first arc in $L(v)$. Otherwise, let $a_i(v) = 0$. Finally, delete each arc in $into(v)$ from the list $L(v)$. This completes the description of *search*.

Edge costs in the directed bridging graph A are computed as follows. Note that the graph induced by the vertex set $V_T(D_i)$ must be connected. Initialize $d(D_i, D_j) = \infty$ for all $i \neq j$. For each vertex v in $V_T(D_i)$ such that v is also in $IP(D_j)$ for some $j \neq i$, add $\langle D_i, D_j\rangle$ to A with cost 0. These are the degenerate bridges. The bridges of length greater than zero are computed as follows. For each edge in the graph induced by $V_T(D_i)$, assign cost zero to that edge. Let v be any vertex in $V_T(D_i)$. Determine the shortest distances $d'(v, w)$ from v to every other vertex w, noting the last vertex v_w in $V_T(D_i)$ on a shortest path to w. Consider each vertex w, where $w \notin V_T(D_i)$ and $w \in IP(D_j)$ for some $j \neq i$. If $d(D_i, D_j)$, the distance from D_i to D_j, is greater than $d'(v, w)$, then update $d(D_i, D_j)$ to be $d'(v, w)$, and identify the corresponding path as (v_w, w).

LEMMA 6. *The directed bridging graph A can be computed in $O(k+qn)$ time, where q is the number of nontrivial components in the balanced graph D.*

Proof. The tree can be rooted at s in $O(n)$ time. The processing time for each component is as follows. With $O(n)$ preprocessing time, the nearest common ancestor for each pair of vertices can be computed in $O(1)$ time [21], [15]. Thus, arcs in D_i can be processed in $O(k_i)$ time, where k_i is the number of objects in D_i. The set of vertices $V_T(D_i)$ can be computed in $O(k_i + n)$ time. With the doubly linked list for $L(v)$ and the address of each arc with terminus v in the list $into(v)$, the deletion of an arc in $L(v)$ can be done in $O(1)$ time. For each vertex in T, the algorithm uses $O(1)$ time to merge the list $L(w)$ into its parent's list, $O(|into(v)|)$ time in deleting arcs from the list $L(v)$ and $O(1)$ time in generating the value of $a_i(v)$. Since T is a tree, the single-source shortest path problem can be solved in $O(n)$ time. The update of the costs on the arcs of A can be done in $O(n)$ time. Therefore, the algorithm runs in $O(k_i + n)$ time for each component of D. Since there are q nontrivial components, the total computation can be implemented in $O(k+qn)$ steps. $\square$

Finally, we show that, given a set of bridges B, the augmented balanced graph can be computed in $O(n)$ time. Let u_i, $1 \leq i \leq r$ be the internal vertices on the path from x_o to y_o at which the arc $\langle x_o, y_o\rangle$ should split. Let $d_1(v)$ be the distance from s to v, in terms of the number of edges. We compute the position $p(u_i, x_o, y_o)$ of vertex u_i, $1 \leq i \leq r$, on arc $\langle x_o, y_o\rangle$ as follows. If u_i is an ancestor of x, then $p(u_i, x_o, y_o)$ is $d_1(x) - d_1(u)$. Otherwise, $p(u_i, x_o, y_o)$ is $d_1(x) + d_1(u_i) - 2d_1(t)$, where t is the nearest common ancestor of x and y. Perform a lexicographic sort on all the triples $(x_o, y_o, p(u, x_o, y_o))$. This sorted list gives, for each arc $\langle x_o, y_o\rangle$, the order for vertices at which the arc $\langle x_o, y_o\rangle$ is to split. Let u'_i, $0 \leq i \leq r$ be such a sequence for $\langle x_o, y_o\rangle$. The arc $\langle x_o, y_o\rangle$ is then replaced by a set of arcs $\langle u'_i, u'_{i+1}\rangle$ $0 \leq i \leq r$, where $u'_0 = x_o$ and $u'_{r+1} = y_o$. These split arcs will have the same label as the original arc, but their costs, which represent distances in T,

will be changed to the corresponding distances that the arcs represent. Since there are q nontrivial components in D, there are at most $q-1$ bridges. Therefore, the augmented graph D_B can be computed in $O(n)$ time.

THEOREM 3. *Given an instance P, let k be the number of objects to be moved and n be the number of vertices in T. The algorithm with-drops can be implemented to compute an optimal transportation for P in $O(k+qn)$ time.*

Proof. The correctness of the algorithm is based on Theorem 2. The balanced graph D can be computed in $O(k+n)$ time. The directed bridging graph A can be computed in $O(k+qn)$ time, and has q vertices. The minimum directed spanning tree of A can be computed in $O(q^2)$ time [22], [12]. The augmented balanced graph can be constructed in $O(n)$ time. Since there are only $O(k+n)$ arcs in D, the generation of the transportation Q can be computed in $O(k+n)$ time. Since $q \leq \min\{k, n\}$, *with-drops* can be implemented to run in $O(k+qn)$ time. □

4. A multilevel approach. In this section, we present another algorithm for our problem. It generates a variation of the bridging graph, called a multilevel bridging graph. This graph is based on a hierarchical decomposition of the tree that produces in general more nodes but fewer directed edges. This allows the directed minimum spanning tree algorithm to run faster in the case that the number of connected components is large as a function of the number of vertices. Our algorithm then runs in $O(k+n\log n)$ time. Thus, it is more efficient asymptotically than the algorithm in the preceding section whenever k is $o(qn)$ and q is $\omega(\log n)$.

We organize this section as follows. First we give a simple transformation for tree T that allows our hierarchical decomposition to be performed efficiently. Then we define our hierarchical decomposition and give an efficient algorithm to find the decomposition. We next specify simple preprocessing of the input that is necessary for generating the multilevel bridging graph. We then describe our algorithm *MULTI_L*, which initializes the multilevel bridging graph and calls a recursive procedure *construct* that adds additional nodes and arcs to the multilevel bridging graph. We carefully analyze the size of the graph generated and the time to generate it. We then show how to extract an optimal solution from a directed minimum spanning tree of the multilevel bridging graph. Finally, we prove correctness and claim the time bound for our algorithm.

Assume that our tree is rooted at s. Our algorithm first uses a clustering approach to transform the tree into a binary tree. Given tree $T_0=(V_0,E_0)$, we shall produce a binary tree $T=(V,E)$. A well-known transformation in graph theory [14, p. 132] is used. For each vertex v with $d>2$ children, $w_1,\cdots,w_d$ and parent w_0, replace v with new vertices $v_1,\cdots,v_{d-1}$. Add edges $\{(v_i,v_{i+1})|i=1,\cdots,d-2\}$, each of cost 0, and replace the edges $\{(v,w_i)|i=1,\cdots,d-1\}$ with edges $\{(v_i,w_i)|i=1,\cdots,d-1\}$, of corresponding costs, and replace the edges $\{(w_0,v),(v,w_d)\}$ with edges $\{(w_0,v_1),(v_{d-1},w_d)\}$, of corresponding costs. The number of vertices and edges will increase by at most $n-3$.

We consider a multilevel approach that generates a different type of bridging graph, which we call a *multilevel bridging graph*. For t a positive integer, let $A^t=(V^t,E^t)$ be the multilevel bridging graph with t levels. For $t>1$, A^t has more nodes than the original one, but has fewer arcs whenever q is $\omega(\log n)$ and k is $o(n\log n)$. Our approach relies on partitioning the tree into clusters. Arcs in the multilevel bridging graph are induced by the subtrees within the clusters, and are also induced by a tree describing the effect of moves across clusters. Our clusters are somewhat similar to, but a variation of, the clusters generated in [7] for a simply-connected topological partition.

Let z be a positive integer to be specified later. Let $T=(V,E)$ be a rooted binary tree. Let the root and at most two other vertices in T be identified as *required boundary*

vertices. Let $E_1, E_2, \cdots, E_l$ be a partition of E. Let the *root* s_i of subgraph $(V(E_i), E_i)$ be the (unique) vertex in $V(E_i)$ nearest the root s of T. An *induced boundary vertex* is a vertex that is in $V(E_i)$ and $V(E_j)$ for some i and some $j \neq i$. An *acceptable clustering* of T of parameter z, $z \geq 2$, is a partition $E_1, E_2, \cdots, E_l$ of E satisfying the following properties:

1. The subgraph $T_i = (V(E_i), E_i)$ is a tree, for $i = 1, \cdots, l$;
2. The number of boundary vertices in $V(E_i) - \{s_i\}$ is at most 2, for $i = 1, \cdots, l$;
3. There are at most $2z - 2$ edges in E_i, for $i = 1, \cdots, l$;
4. There are at most three sets E_i such that there are both fewer than z edges in E_i and fewer than 3 boundary vertices in $V(E_i)$.

Each subgraph T_i is called a *cluster*.

Clusters can be generated as follows. Recursive procedure *cl_search* is called with parameters s and z. Procedure *cl_search*(v, z) partitions the edges in the subtree rooted at v into zero or more clusters, and one set of at most $z-1$ edges. The clusters with their boundary vertices are output, and the set of remaining edges, with its boundary vertices, are returned to the calling procedure.

When *cl_search* is called with parameters v and z, the following is done. A cluster C and a set BV of boundary vertices are both initialized to the empty set. If v is both a leaf and a required boundary vertex, then v is inserted into BV, and C and BV are returned to the calling procedure. Otherwise, *cl_search* does the following. First, for each child w of v, edge (v, w) is inserted into C, *cl_search*(w, z) is called and returns C' and BV', and C' and BV' are unioned into C and BV, respectively. Second, if $|C| \geq z$ or $|BV| = 2$ or v is a required boundary vertex, then v is inserted into BV, cluster C is printed, along with boundary vertices BV and root v, and C is reset to be empty and BV to be $\{v\}$. Third, C and BV are returned to the calling procedure.

Let a procedure *FINDCL* be the procedure that initially calls *cl_search* with parameters s and z, and let (C, BV) be returned to *FINDCL*. The set C will be empty, since s is a required boundary vertex.

LEMMA 7. *Let T be a rooted binary tree of m edges. Let z be a positive integer, $z \geq 2$. The number of clusters in an acceptable clustering of T of parameter z is at most $1 + (2m + 2)/(z + 2)$.*

Proof. There are at most 3 clusters that contain required boundary vertices, and these may contain as few as one edge. We count the remaining l' clusters as follows. From among these l' clusters, let n_i be the number of clusters with i boundary vertices for $i = 1, 2, 3$. First note that $n_1 + n_2 + n_3 = l'$. These clusters induce a tree T', where the nodes correspond to clusters, and there is an edge between two clusters if they share a vertex in T. Thus $n_1 + 2n_2 + 3n_3 = 2(l' - 1)$ follows by noting that total degree in T' is twice the number of edges. From these equations we infer that $n_1 = n_3 + 2$.

Each of the n_3 clusters will contain at least two edges, and each of the $n_1 + n_2$ clusters will contain at least z edges. Thus $3 + 2n_3 + z(n_1 + n_2) \leq m$, which implies that $3 + 2n_3 + z(n_3 + 2) \leq m$. Thus $n_3 \leq ((m - 3 - 2z)/(z + 2))$. It also follows that $n_1 + n_2 \leq (m - 3 - 2n_3)/z$. Thus the total of all clusters is $3 + n_1 + n_2 + n_3$, which will be at most $3 + (m - 3 - 2n_3)/z + n_3$, which is at most $3 + (m-3)z + (1 - 2/z)(m - 3 - 2z)/(z + 2)$, which equals the claimed bound, as long as $z \geq 2$. □

Considering the tree T in Fig. 3, we give an acceptable clustering of T of parameter $z = 2$ in Fig. 5. The clusters formed will have edge sets $E_1 = \{(0, 1)\}$, $E_2 = \{(0, 2), (2, 4)\}$, $E_3 = \{(2, 3), (3, 5)\}$, $E_4 = \{(5, 7), (5, 8)\}$, and $E_5 = \{(3, 6), (6, 9)\}$. Note that each of the five subgraphs identified is a tree. The roots of T_1, T_2, T_3, T_4, and T_5 are 0, 0, 2, 5, and 3, respectively. Tree T_1 has one boundary vertex, vertex 0; tree T_2 has two

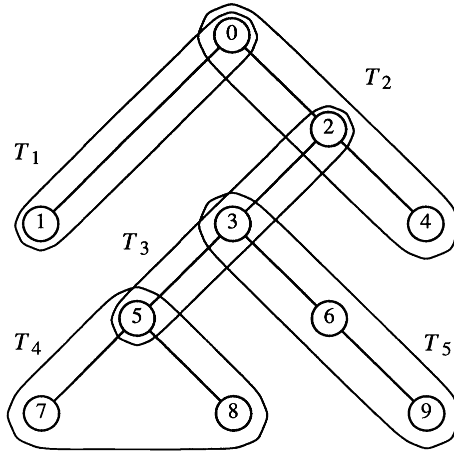


FIG. 5. *Clusters of the tree in Fig.* 3.

boundary vertices, 0 and 2; tree T_3 has three boundary vertices, 2, 3 and 5; tree T_4 has one boundary vertex, vertex 5; and tree T_5 has one boundary vertex, vertex 3. Each tree has $z = 2 = 2z - 2$ edges except for tree T_1, which has one edge.

We next specify the preprocessing of the input for the routine that creates the multilevel bridging graph. The preprocessing involves finding a reduced set of moves of size $O(n)$. The reduced set satisfies the property that for each component D_i, every vertex in $V_T(D_i)$ is covered by a move in the reduced set. For each component D_i do the following. First, choose some vertex v in $IP(D_i)$. Second, find a depth-first search tree of D_i rooted at v. Third, find a spanning tree of D_i rooted at v with arcs directed toward v. This can be done by reversing the direction of all arcs in D_i, finding a depth-first search tree rooted at v, and then reversing the direction of the arcs back to what they were. Fourth, union the arcs of the two trees together. This gives a set of arcs that are strongly connected, and of size proportional to the cardinality of $IP(D_i)$, in time proportional to the number of arcs in D_i. The union of these sets over all components D_i is the reduced set of moves used in generating the multilevel bridging graph. Clearly, the set of reduced moves is of size $O(n)$, and can be found in $O(k)$ time.

We assume that the input to our algorithms to construct bridging graphs is in the following form. First there is a weighted rooted binary tree T, of which the root and at most two other vertices are designated as required boundary vertices. Second is a reduced set M of moves $\langle x, y\rangle$ with each having a label indicating which component D_i it is a member of, along with the original name for the move. Since we will be using a multilevel approach, within the recursion these moves may represent portions of moves in the original problem. Thus in our input two moves with different component labels may be incident on the same vertex. We avoid problems by assuming that in the recursion no moves have their endpoints treated as initial positions.

We next describe our algorithm *MULTI_L*, which constructs a multilevel bridging graph $A^t = (V^t, E^t)$. Algorithm *MULTI_L* initializes V^t and E^t as follows. For each component D_i, $i = 1, 2, \cdots, q$, a node $\bar{i}$ is inserted into V^t. For each vertex v in T, $v = 0, 1, \cdots, n-1$, a node v is inserted into V^t. Thus initially there are $q + n$ nodes in V^t. Initialize *label_count* to q. For each component D_i, $i = 1, 2, \cdots, q$, for each vertex v in $IP(D_i)$, insert arcs $\langle v, \bar{i}\rangle$ and $\langle \bar{i}, v\rangle$ into E^t with cost 0, *orig_name* set to *null*, and

drop set to v. For each edge (u, v) in T, with u closer to the start vertex than v, insert $\langle u, v \rangle$ into E^t with cost $c(u, v)$, *orig_name* set to *null*, and *drop* set to u. If the start vertex s is in a component D_i by itself, insert arc $\langle \bar{i}, s \rangle$ into E^t with cost 0, *orig_name* set to *null*, and *drop* set to s. Thus there are initially at most $3n$ arcs in E^t. Algorithm *MULTI_L* next sets up the set M of moves as follows. Each move $\langle x, y \rangle$ in M has a label i for component D_i containing $\langle x, y \rangle$, and *orig_name* set to "$\langle x, y \rangle$." Let the moves in M be ordered by label value. *MULTI_L* then calls a recursive procedure *construct*, which identifies additional nodes and arcs of A^t. The procedure is invoked with tree T, reduced set M of moves, and an appropriate number of levels t, which we identify later.

We now discuss our recursive procedure *construct*, which is called with parameters T, M, and t, where t is a positive integer. Let m be the number of edges in T. We choose a suitable constant $m_0 = 15$ at which to stop the recursion. If $t = 1$ or $m \leq m_0$, then *construct* handles T and M somewhat similarly to the approach in §3. We shall specify this carefully after seeing how the recursion is handled.

If $t > 1$ and $m > m_0$, the following is done. Let $z = \lfloor m^{1-1/t}/2 \rfloor$. Note that by the conditions on t and m, $z \geq 2$. An acceptable clustering $T_1, \cdots, T_l$ of tree T for parameter z is found. If $l = 1$, then recursively invoke procedure *construct* with arguments T, M, and $t - 1$. Otherwise, if $l > 1$, then do the following. Define the *compressed tree* $\overline{T}$ as follows. Initialize $\overline{T}$ as a copy of T. Next delete all vertices in T not on a path between two boundary vertices. Then while there is a nonboundary vertex v of degree 2, replace v and its two adjacent edges (u, v) and (v, w) by the edge (u, w) of cost $c(u, v) + c(v, w)$. Tree $\overline{T}$ is the resulting tree. Associated with each edge in $\overline{T}$ is a path in T, which we call a *basic path*. There are at most three basic paths in any one cluster. In the case that there are three, they all share one nonboundary vertex as an endpoint.

We define a set $\overline{M}$ of moves for $\overline{T}$, and sets M_i of moves for tree T_i in the clustering, $i = 1, \cdots, l$, as follows. For each move $\langle x, y \rangle$ in M, do the following. Suppose that the label of $\langle x, y \rangle$ is i. If there is a cluster T_j such that both x and y are in $V(E_j)$, then insert $\langle x, y \rangle$ into M_j with label i and the same value of *orig_name*. Otherwise do the following. Let u and v be boundary vertices on the path from x to y in T such that the path from u to v in T is of maximum length. If $u \neq x$, then x is in only one cluster T_j, and move $\langle x, u \rangle$ is inserted into M_j with label i and with $orig_name(x, u) = orig_name(x, y)$. If $v \neq y$, then y is in only one cluster $T_{j'}$, and move $\langle v, y \rangle$ is inserted into $M_{j'}$ with label i and with $orig_name(v, y) = orig_name(x, y)$. If $u \neq v$, insert $\langle u, v \rangle$ into $\overline{M}$, with label i and with $orig_name(u, v) = orig_name(x, y)$. This completes the description of how to handle each move $\langle x, y \rangle$. Since M can be examined in order of label value, the moves in $\overline{M}$ and in the sets M_i for $i = 1, 2, \cdots, l$ are generated in order of label value.

Whenever endpoint x or y of a move $\langle x, y \rangle$ is not a boundary vertex, then the corresponding vertex u or v can be identified in constant time as follows. Assume that preorder and postorder numbers have been computed for T, so that ancestor testing can be done in constant time. Let s_j be the root of T_j, and let s'_j be the root of T'_j. If s_j is an ancestor of s'_j, then choose u as the boundary vertex in T_j that is an ancestor of y and a proper descendant of s_j, and choose v as s'_j. A similar approach applies if s'_j is an ancestor of s_j. If neither s_j nor s'_j is an ancestor of the other, choose u as s_j and v as s'_j.

The construction of M_j, $j = 1, \cdots, l$ is completed as follows. Determine which edges in $\overline{T}$ are covered by moves in $\overline{M}$. For each such edge $e = (y_1, y_2)$, do the following. Increment *label_count*, and insert a node with index $\bar{i}$, where $i = label_count$, into V^t. These nodes can be viewed as transfer nodes: a number of different components may have moves that cover edge e, but in M_j these will all be represented by one move with label i. Let T_j be the cluster containing both y_1 and y_2. Insert move $\langle y_1, y_2 \rangle$ into M_j

with label i and *orig_name* set to *null*. Note that moves in M_j are still ordered by label value.

The processing of $\overline{T}$ is completed by generating additional arcs for E^t from $\overline{T}$ as follows. For each label i of moves in $\overline{M}$ do the following. Determine the set of edges e in $\overline{T}$ such that there is a move with label i in $\overline{M}$ that covers e. For each such edge e, do the following. Choose some move $\langle x, y\rangle \in \overline{M}$ with label i that covers $e = (y_1, y_2)$, and let the node for e in V^t have index i'. Insert arc $\langle \bar{i}, \bar{i'}\rangle$ into E^t with cost 0, label equal to the label of move $\langle x, y\rangle$, and drop set to $null$.

For each cluster T_j, recursively invoke our procedure *construct* with arguments T_j, M_j and $t-1$. This invocation will add some number of arcs and nodes to A^t. This completes the description of the recursion step of *construct*.

We now discuss how *construct* handles the case when $t = 1$ or $m \leq m_0$. Consider the set of labels for moves in M. For each label i, do the following. Determine all vertices covered by moves with this label. For each such vertex u, if there is a move with label i and nonnull original name that covers u, let $a_i(u)$ be the original name of such a move. For every vertex u covered by a move with label i, insert into E^t an arc $\langle \bar{i}, u\rangle$ with cost 0, *orig_name* set to $a_i(u)$, and *drop* set to u. This completes the description of the basis case of *construct*, and with it the description of all of procedure *construct*.

We illustrate algorithm *MULTI_L* and procedure *construct* with an example. We take $t = 2$ and construct a 2-level bridging graph for the tree T and set of moves M shown in Fig. 3. The resulting 2-level bridging graph is shown in Fig. 6. Algorithm *MULTI_L* initializes V^t with nodes $\bar{1}, \bar{2}, \bar{3}, \bar{4}$ representing components D_1, D_2, D_3, D_4, and nodes $0, 1, \cdots, 9$ representing vertices $0, 1, \cdots, 9$ in T. Also, *MULTI_L* initializes E^t with arcs, which we shall designate with quadruples $(\langle \bar{i}, v\rangle, c, o_n, d)$, where $\langle \bar{i}, v\rangle$ is an arc with cost c, original name o_n, and drop vertex d. The arcs from nodes representing components to nodes representing initial positions are $(\langle \bar{2}, 1\rangle, 0, null, 1)$, $(\langle 1, \bar{2}\rangle, 0, null, 1)$, $(\langle \bar{2}, 7\rangle, 0, null, 7)$, $(\langle 7, \bar{2}\rangle, 0, null, 7)$, $(\langle \bar{3}, 5\rangle, 0, null, 5)$, $(\langle 5, \bar{3}\rangle, 0, null, 5)$, $(\langle \bar{3}, 8\rangle, 0, null, 8)$, $(\langle 8, \bar{3}\rangle, 0, null, 8)$, $(\langle \bar{4}, 4\rangle, 0, null, 4)$, $(\langle 4, \bar{4}\rangle, 0, null, 4)$, $(\langle \bar{4}, 6\rangle, 0, null, 6)$, $(\langle 6, \bar{4}\rangle, 0, null, 6)$, $(\langle \bar{4}, 9\rangle, 0, null, 9)$, and $(\langle 9, \bar{4}\rangle, 0, null, 9)$. The arcs corresponding to edges in T are $(\langle 0, 1\rangle, 9, null, 0)$, $(\langle 0, 2\rangle, 1, null, 0)$, $(\langle 2, 3\rangle, 1, null, 2)$, $(\langle 2, 4\rangle, 1, null, 2)$, $(\langle 3, 5\rangle, 1, null, 3)$, $(\langle 3, 6\rangle, 1, null, 3)$, $(\langle 5, 7\rangle, 1, null, 5)$, $(\langle 5, 8\rangle, 1, null, 5)$, and $(\langle 6, 9\rangle, 1, null, 6)$. We shall designate the moves in M by a triple $(\langle u, v\rangle, i, o_n)$, where i is the label and o_n is the original name of some move. Thus M consists of $(\langle 1, 7\rangle, 2,$ "$\langle 1, 7\rangle$"$)$, $(\langle 7, 1\rangle, 2,$ "$\langle 7, 1\rangle$"$)$, $(\langle 5, 8\rangle, 3,$ "$\langle 5, 8\rangle$"$)$, $(\langle 8, 5\rangle, 3,$ "$\langle 8, 5\rangle$"$)$, $(\langle 4, 9\rangle, 4,$ "$\langle 4, 9\rangle$"$)$, $(\langle 6, 4\rangle, 4,$ "$\langle 6, 4\rangle$"$)$, and $(\langle 9, 6\rangle, 4,$ "$\langle 9, 6\rangle$"$)$.

For the sake of our example, we shall assume that $z = 2$. (Actually, the smallest value of m for which $z = 2$ would be $m = 16$, but considering a tree of this size would unnecessarily clutter the example.) We use the clusters as shown in Fig. 5. The boundary vertices, besides the root (vertex 0), will be 2, 3, 5. Thus the compressed tree $\overline{T}$ contains edges $(0, 2)$, $(2, 3)$, and $(3, 5)$.

We generate $\overline{M}$ and sets M_i from M as follows. For $(\langle 1, 7\rangle, 2,$ "$\langle 1, 7\rangle$"$)$, we insert $(\langle 1, 0\rangle, 2,$ "$\langle 1, 7\rangle$"$)$ into M_1, $(\langle 5, 7\rangle, 2,$ "$\langle 1, 7\rangle$"$)$ into M_4, and $(\langle 0, 5\rangle, 2,$ "$\langle 1, 7\rangle$"$)$ into $\overline{M}$. For $(\langle 7, 1\rangle, 2,$ "$\langle 7, 1\rangle$"$)$, we insert $(\langle 7, 5\rangle, 2,$ "$\langle 7, 1\rangle$"$)$ into M_4, $(\langle 0, 1\rangle, 2,$ "$\langle 7, 1\rangle$"$)$ into M_1, and $(\langle 5, 0\rangle, 2,$ "$\langle 7, 1\rangle$"$)$ into $\overline{M}$. For $(\langle 5, 8\rangle, 3,$ "$\langle 5, 8\rangle$"$)$, we insert $(\langle 5, 8\rangle, 3,$ "$\langle 5, 8\rangle$"$)$ into M_4. For $(\langle 8, 5\rangle, 3,$ "$\langle 8, 5\rangle$"$)$, we insert $(\langle 8, 5\rangle, 3,$ "$\langle 8, 5\rangle$"$)$ into M_4. For $(\langle 4, 9\rangle, 4,$ "$\langle 4, 9\rangle$"$)$, we insert $(\langle 4, 2\rangle, 4,$ "$\langle 4, 9\rangle$"$)$ into M_2, $(\langle 3, 9\rangle, 4,$ "$\langle 4, 9\rangle$"$)$ into M_5, and $(\langle 2, 3\rangle, 4,$ "$\langle 4, 9\rangle$"$)$ into $\overline{M}$. For $(\langle 6, 4\rangle, 4,$ "$\langle 6, 4\rangle$"$)$, we insert $(\langle 6, 3\rangle, 4,$ "$\langle 6, 4\rangle$"$)$ into M_5, $(\langle 2, 4\rangle, 4,$ "$\langle 6, 4\rangle$"$)$ into M_2, and $(\langle 3, 2\rangle, 4,$ "$\langle 6, 4\rangle$"$)$ into $\overline{M}$. For $(\langle 9, 6\rangle, 4,$ "$\langle 9, 6\rangle$"$)$, we insert $(\langle 9, 6\rangle, 4,$ "$\langle 9, 6\rangle$"$)$ into M_5.

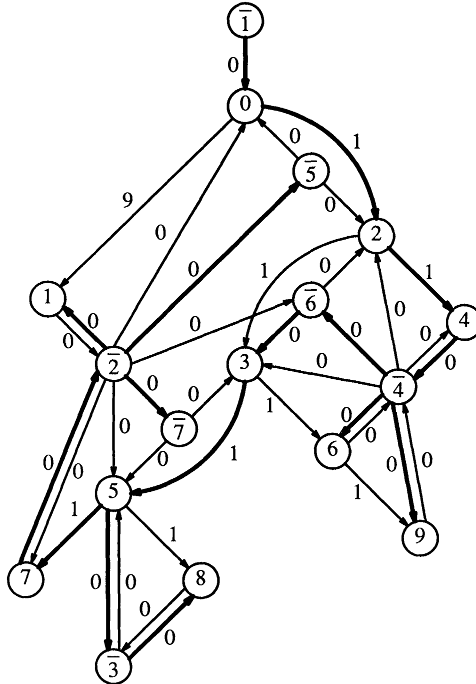


FIG. 6. *The multilevel bridging graph for Fig.* 3, *with a directed minimum spanning tree in bold.*

For the compressed tree $\overline{T}$, we create nodes and arcs as follows. Create and insert into V^t node $\overline{5}$ for edge $(0, 2)$, node $\overline{6}$ for edge $(2, 3)$, and node $\overline{7}$ for edge $(3, 5)$. Also, insert into E^t($\langle\overline{2}, \overline{5}\rangle$, 0,"$\langle 1, 7\rangle$", *null*), ($\langle\overline{2}, \overline{6}\rangle$, 0, "$\langle 1, 7\rangle$", *null*), ($\langle\overline{2}, \overline{7}\rangle$, 0, "$\langle 1, 7\rangle$", *null*), and ($\langle\overline{4}, \overline{6}\rangle$, 0, "$\langle 4, 9\rangle$", *null*).

Next, *construct* is applied to each cluster T_j. Considering M_1, *construct* sets $a_2(1) = \langle 1, 7\rangle$, $a_2(0) = \langle 1, 7\rangle$, and inserts into E^t ($\langle\overline{2}, 0\rangle$, 0, "$\langle 1, 7\rangle$," 0), and ($\langle\overline{2}, 1\rangle$, 0, "$\langle 1, 7\rangle$," 1). Considering M_2, *construct* sets $a_4(2) = \langle 4, 9\rangle$, $a_4(4) = \langle 4, 9\rangle$, $a_5(0) = null$, and $a_5(2) = null$, and inserts into E^t($\langle\overline{4}, 2\rangle$, 0, "$\langle 4, 9\rangle$," 2), ($\langle\overline{4}, 4\rangle$, 0, "$\langle 4, 9\rangle$," 4), ($\langle\overline{5}, 0\rangle$, 0, *null*, 0), and ($\langle\overline{5}, 2\rangle$, 0, *null*, 2). Considering M_3, *construct* sets $a_6(2) = null$, $a_6(3) = null$, $a_7(3) = null$, and $a_7(5) = null$, and inserts into E^t ($\langle\overline{6}, 2\rangle$, 0, "*null*," 2), ($\langle\overline{6}, 3\rangle$, 0, "*null*," 3), ($\langle\overline{7}, 3\rangle$, 0, "*null*," 3), and ($\langle\overline{7}, 5\rangle$, 0, "*null*," 5). Considering M_4, *construct* sets $a_2(5) = \langle 1, 7\rangle$, $a_2(7) = \langle 1, 7\rangle$, $a_3(5) = \langle 5, 8\rangle$, and $a_3(8) = \langle 5, 8\rangle$, and inserts into E^t ($\langle\overline{2}, 5\rangle$, 0, "$\langle 1, 7\rangle$", 5), ($\langle\overline{2}, 7\rangle$, 0, "$\langle 1, 7\rangle$", 7), ($\langle\overline{3}, 5\rangle$, 0, "$\langle 5, 8\rangle$", 5), and ($\langle\overline{3}, 8\rangle$, 0, "$\langle 5, 8\rangle$", 8). Considering M_5, *construct* sets $a_4(3) = \langle 4, 9\rangle$, $a_4(6) = \langle 9, 6\rangle$, and $a_4(9) = \langle 9, 6\rangle$, and inserts into E^t($\langle\overline{4}, 3\rangle$, 0, "$\langle 4, 9\rangle$", 3), ($\langle\overline{4}, 6\rangle$, 0, "$\langle 9, 6\rangle$", 6), and ($\langle\overline{4}, 9\rangle$, 0, "$\langle 9, 6\rangle$", 9).

This completes the construction of the multilevel bridging graph A^t for our example. We note that for this particular example A^t is considerably larger than the regular bridging graph A. This is due to our example being relatively small, and the number of components being relatively small.

We next analyze the time requirements of procedure *construct*, and the number of nodes and arcs added by it to A^t.

LEMMA 8. *Let T be a weighted rooted binary tree with m edges and at most* 3 *required boundary vertices. Let M be a set of k moves with $q \leq m + 4$ different labels. Let d be the total number of endpoints of moves in M that are not required boundary vertices. Let t be a positive integer. Procedure construct uses $O(t(k + m^{1+1/t}))$ time. The number of nodes*

and arcs introduced into the multilevel directed bridging graph A^t by procedure construct are $O(m)$ and $O(t(k + m^{1+1/t}))$, respectively.

Proof. Suppose that $t = 1$ or $m \leq m_0$. Then no nodes are inserted into V^t by *construct*. For each label value i, determining the set of vertices covered by moves with label i, contracting the tree, finding shortest distances from v_i', and generating the arcs reflecting these shortest distances can be performed in $O(k_i + m)$ time, where k_i is the number of moves with label i. Since there are $q \leq m + 4$ different labels, the time used is $O(k + m^2)$. It can easily be seen that at most $(m + 4)(m + 3)$ edges are generated in this case.

Suppose that $t > 1$ and $m > m_0$. By Lemma 7, at most $1 + (2m + 2)/(z + 2) < 1 + (2m+2)/(m^{1-1/t}/2+3/2) = 1+(4m+4)/(m^{1-1/t}+3) < 1+(4m+12m^{1/t})/(m^{1-1/t}+3) = 1 + 4m^{1/t}$ clusters are created. Since all but at most 4 vertices in any cluster are deleted when $\overline{T}$ is generated, $\overline{T}$ has fewer than $4 + 16m^{1/t}$ vertices, and thus fewer than $3 + 16m^{1/t}$ edges. Thus *construct* generates $O(m^{1/t})$ nodes in handling $\overline{T}$. It also generates $O(q\overline{m})$ edges, where $\overline{m}$ is the number of edges in $\overline{T}$. Thus *construct* generates $O(m^{1+1/t})$ arcs, and uses $O(k + m^{1+1/t})$ time in handling $\overline{T}$. The time required to set up $\overline{T}$, $\overline{M}$ and M_j, $j = 1, 2, \cdots, l$, and to handle $\overline{T}$ is $O(k + m^{1+1/t})$.

Let m_j be the number of edges in cluster C_j. From the clustering method, it follows that $\sum_{j=1}^{l} m_j = m$. By choice of parameter z, $m_j \leq m^{1-1/t}$. Let q_j be the number of labels of moves for tree T_j. We bound q_j as follows. There are $m_j + 1$ vertices in T_j, each of which can be an initial position for a different component. In addition, there can be one component for each of at most three basic paths in T_j. Thus there can be a total of at most $m_j + 4$ components in T_j.

We first analyze the number of arcs introduced by *construct* into E^t. Let $R(m, t)$ be the number of arcs introduced by *construct* for a tree with m edges, and with moves of at most $m + 4$ different labels, and parameter t. From the above discussion, $R(m, t)$ is bounded by the recurrence:

$$R(m,t) \leq \begin{cases} cm^2, & \text{for } t = 1 \quad \text{or} \quad m \leq m_0; \\ cm^{1+1/t} + \sum_{j=1}^{l} R(m_j, t-1) & \text{for } t > 1 \quad \text{and} \quad m > m_0, \end{cases}$$

where c is an appropriate constant.

We claim that $R(m, t) \leq ctm^{1+1/t}$. The proof is by induction on t. The basis, with $t = 1$, follows immediately, since *construct* generates $O(m^2)$ arcs. For the induction step, with $t > 1$, assume as the induction hypothesis that the claim is true for $t - 1$. Then we have

$$\begin{aligned} R(m,t) &\leq cm^{1+1/t} + \textstyle\sum_{j=1}^{l} R(m_j, t-1) \\ &\leq cm^{1+1/t} + \textstyle\sum_{j=1}^{l} c(t-1)m_j^{1+1/(t-1)}, \end{aligned}$$

by the induction hypothesis. The sum is maximized when the values for m_j are as large as possible, i.e., $2z - 2$. Thus

$$
\begin{aligned}
R(m,t) &\le cm^{1+1/t} + c(t-1)(2z-2)^{1+1/(t-1)}m/(2z-2) \\
&< cm^{1+1/t} + c(t-1)(2z)^{1/(t-1)}m \\
&\le cm^{1+1/t} + c(t-1)(m^{1-1/t})^{1/(t-1)}m \\
&= ctm^{1+1/t}.
\end{aligned}
$$

We next bound the number of additional nodes introduced into the multilevel bridging graph by *construct* for a tree with m edges. We count the number of nodes resulting from basic paths. If $t = 1$ or $m \le m_0$, then no new nodes are introduced. Otherwise, the tree is partitioned into l clusters. If $l = 1$, then no new nodes are created, but the procedure is called recursively. If $l > 1$, then a node is introduced for each basic path, of which there at most 3 per cluster. Then the procedure is called recursively on each cluster. Consider a decomposition tree of the original tree T, where the root represents tree T, and every other node represents a cluster generated at some point by *construct*. Each node representing a cluster T' has as its children nodes representing the clusters that T' is partitioned into. The number of leaves in the decomposition tree is less than or equal to m. The total number of children of all nodes with at least 2 children is less than $2m$. Thus the total number of nodes added to the multilevel directed bridging graph because of basic paths will be less than $6m$.

We next analyze the time used by *construct*. Let $T(m, k, k', t)$ be the number of arcs introduced by *construct* for a tree with m edges, k moves, of which k' have both endpoints not being boundary vertices, and with at most $m + 4$ different labels, and parameter t. Note that the number of vertices of degree 1 or 2 in T is at least $(m + 3)/2$. Since these vertices would have been deleted if they were not initial positions or destinations, we must have $k \ge (m + 3)/4$. For $m > m_0 = 15$, $k \ge 5$, and thus there is a constant c such that the time to handle $\overline{T}$ is at most $c(k - 3 + m^{1+1/t})$.

Let k_j be the number of moves in the problem for cluster T_j, and let k'_j be the number of these moves having both endpoints not being boundary vertices. In generating the problems for the clusters T_j, some moves for T may be split. If a move in T already has at least one endpoint that is a boundary vertex, then there can be a corresponding move in at most one cluster T_j. If a move in T already has both endpoints not being boundary vertices, if the move is split, then it is replaced by two moves in the clusters, each of which has at least one endpoint that is a boundary vertex. Also, each cluster can receive at most three new moves, corresponding to basic paths. Thus $\sum_{j=1}^{l} k_j \le k + (k' - \sum_{j=1}^{l} k'_j) + 3l$. It follows that the function $T(m, k, k', t)$ is bounded by the recurrence:

$$
T(m,k,k',t) \le \begin{cases} c(k+m^2), & \text{for } t = 1 \quad \text{or} \quad m \le m_0; \\ c(k-3+m^{1+1/t}) + \sum_{j=1}^{l} T(m_j, k_j, k'_j, t-1) & \text{for } t > 1 \quad \text{and} \quad m > m_0, \end{cases}
$$

where c is an appropriate constant.

We claim that $T(m, k, k', t) \le ct(k + k' - 3 + m^{1+1/t})$. The proof is by induction on t. The basis, with $t = 1$, follows immediately. For the induction step, with $t > 1$, assume

as the induction hypothesis that the claim is true for $t-1$. Then we have

$$\begin{aligned} &T(m,k,k',t)\\ &\le c(k-3+m^{1+1/t})+\sum_{j=1}^{l} T(m_j,k_j,k_j',t-1)\\ &\le c(k-3+m^{1+1/t})+\sum_{j=1}^{l} c(t-1)(k_j+k_j'-3+m^{1+1/(t-1)}), \end{aligned}$$

by the induction hypothesis. The sum is maximized when the values for m_j are as large as possible, i.e., $2z-2$. Thus

$$\begin{aligned} &T(m,k,k',t)\\ &\le c(k-3+m^{1+1/t})+c(t-1)\left(\sum_{j=1}^{l} k_j+\sum_{j=1}^{l} k_j'-3l+(2z-2)^{1+1/(t-1)}m/(2z-2)\right)\\ &< c(k-3+m^{1+1/t})+c(t-1)((k+k'-\sum_{j=1}^{l} k_j'+3l)+\sum_{j=1}^{l} k_j'-3l+(2z)^{1/(t-1)}m)\\ &\le c(k-3+m^{1+1/t})+c(t-1)(k+k'+(m^{1-1/t})^{1/(t-1)}m)\\ &\le ct(k+k'-3+m^{1+1/t}) \end{aligned}$$

This concludes the proof. □

Upon the return of *construct* to algorithm *MULTI_L*, all nodes and a multiset of arcs of the multilevel bridging graph have been identified. A 2-pass radix sort is then performed to sort the arcs lexicographically, and eliminate all but the least expensive of multiple arcs. For each such arc the label and *orig_name* are brought along. The directed minimum spanning tree algorithm of [12] can then be applied to find a directed minimum spanning tree rooted at the node corresponding to the component that contains the start vertex.

The directed minimum spanning tree of A^t can be translated into a minimum cost set of bridges as follows. There will be a bridge in the set from component D_i to component D_j if and only if there is a directed path in the directed minimum spanning tree to node $\overline{j}$ from either node $\overline{i}$ or an initial position in D_i. that contains no intermediate node with index in $\{\overline{1},\overline{2},\cdots,\overline{q}\}$. The first arc in this directed path carries in its *orig_name* field the name of a move that should be interrupted. (If *orig_name* is *null*, or if the *drop* location is an endpoint of the move, then no move is interrupted, implying that the bridge starts at an initial position.) The first arc on this path that enters a node with index in $\{0,1,\cdots,n-1\}$ will contain in *drop* the name of the vertex at which to drop the object. Once these values are known, the transportation can be constructed as in §3.

We return to our running example. A directed minimum spanning tree is shown in bold for our multilevel bridging graph in Fig. 6. Since there are paths $P_1=\overline{1},0,2,4,\overline{4}$, $P_2=\overline{4},\overline{6},3,5,\overline{3}$, and $P_3=5,7,\overline{2}$ in the directed minimum spanning tree, there are bridges from D_1 to D_4, D_4 to D_3, and D_3 to D_2. The first arcs on each of these paths that enter a node with index in $\{0,1,\cdots,n-1\}$ are $\langle\overline{1},0\rangle$, $\langle\overline{6},3\rangle$, and $\langle 5,7\rangle$, respectively. These arcs have drop field equal to 0, 3, and 5, respectively. Thus the first bridge is from 0 to 4, the second from 3 to 5, and the third from 5 to 7. The first arcs on paths P_1, P_2,

and P_3 are $\langle \bar{1}, 0\rangle$, $\langle \bar{4}, \bar{6}\rangle$, $\langle \bar{3}, 7\rangle$, respectively. These arcs have original name field equal to *null*, $\langle 4, 9\rangle$ and $\langle 5, 8\rangle$, respectively. It follows that no move is interrupted for the first bridge. Since 3 is not an endpoint of $\langle 4, 9\rangle$, move $\langle 4, 9\rangle$ is interrupted at 3 for the second bridge. Since one of the endpoints of $\langle 5, 8\rangle$ is the drop value, no move is interrupted for the third bridge. We note that we could just as well have chosen for P_1 the path $0, 2, 4, \bar{4}$, since 0 is an initial position in D_1. Since the drop field of arc $\langle 0, 2\rangle$ is 0, and the original name of $\langle 0, 2\rangle$ is *null*, no change would result.

THEOREM 4. *Let T be a weighted rooted binary tree with n vertices for which there is a set of k moves. Algorithm MULTI_L finds a minimum-cost preemptive transportation in $O(k + n \log n)$ time.*

Proof. We first address correctness. First we claim that the arcs that comprise any bridge are contained in the multilevel bridging graph A^t. Clearly, for every tree edge an arc is introduced, so that the only issue is whether the correct direction is chosen for the arc. But in any traversal of the tree, the first time any given edge is traversed, it will be traversed in the direction away from the start vertex.

Next we claim that for any initial position u of a component D_i, and any vertex v in $V_T(D_i)$, there is a corresponding path in A^t from u to v of cost 0. First note that by introducing arcs from $\bar{i}$ to each initial position in D_i, and, vice versa, there is a path of cost 0 in A^t between any pair of initial positions in D_i. Next consider the move $\langle x, y\rangle$ in the reduced set of moves that covers vertex v in $V_T(D_i)$. It can be shown by induction on t that there is a path $\bar{i} = \overline{i_0}, \overline{i_1}, \cdots, \overline{i_p}, v$ of cost 0. Thus there is a directed minimum spanning tree of A^t of cost equal to the cost of a minimum-cost set of bridges.

Finally, we verify that the appropriate information is associated with each arc in A^t. Whenever a move is split in our construction of A^t, the original name of the move is retained. Furthermore, in generating an arc $\langle \bar{i}, u\rangle$ to indicate that a move with label i covers vertex u, *drop*$(\bar{i}, u)$ is set to u. This completes the discussion of correctness.

We next discuss the time used by our algorithm. Let t be a positive integer. By Lemma 8, the time to generate V^t and a multiset containing E^t is $O(t(n + n^{1+1/t}))$, assuming that the reduced set of moves is used, so that k is $O(n)$, and noting that the number of edges is $n - 1$. Multiple arcs can be deleted in $O(tn^{1+1/t})$ time. Since the algorithm of [12] uses $O(m + n \log n)$ time on a graph of n nodes and m arcs, the time to find a directed minimum spanning tree is $O(tn^{1+1/t} + n \log n)$. The time to translate a directed minimum spanning tree of A^t into a set of bridges with drop points specified is proportional to the size of the tree. The time to generate the transportation, given this information is $O(k + n)$. Therefore, the algorithm *MULTI_L* can be implemented to run in $O(k + tn^{1+1/t} + n \log n)$ time. Choosing $t = \log n$ yields a running time of $O(k + n \log n)$. □

Acknowledgment. We would like to thank the referee for his helpful comments.

REFERENCES

[1] M. J. ATALLAH AND S. R. KOSARAJU, *Efficient solutions to some transportation problems with application to minimizing robot arm travel*, SIAM J. Comput., 17 (1988), pp. 849–869.

[2] F. BOCK, *An algorithm to construct a minimum directed spanning tree in a directed network*, in Developments in Operations Research; Proceedings of the Third Annual Israel Conference on Operations Research, B. Avi-Itzhak, ed., Gordon and Breach, New York, 1971, pp. 29–44.

[3] J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, The MacMillan Press, London, England, 1976.

[4] P. M. CAMERINI, L. FRATTA, AND F. MAFFIOLI, *A note on finding optimum branchings*, Networks, 9 (1979), pp. 309–312.

[5] Y. J. CHU AND T. H. LIU, *On the shortest arborescence of a directed graph*, Scientia Sinica, 14 (1965), pp. 1396–1400.

[6] J. EDMONDS, *Optimum branchings*, J. Res. Nat. Bureau of Standards—B. Math. and Math. Phys., 71B (1976), pp. 233–240.

[7] G. N. FREDERICKSON, *Data structures for on-line updating of minimum spanning trees, with applications*, SIAM J. Comput., 14 (1985), pp. 781–798.

[8] ———, *A note on the complexity of a simple transportation problem*, SIAM J. Comput., 22 (1993), to appear.

[9] G. N. FREDERICKSON AND D. J. GUAN, *Ensemble motion planning with a vehicle of capacity greater than one*, manuscript, 1989.

[10] ———, *Nonpreemptive ensemble motion planning on a tree*, Tech. Report CSD-TR-864, Department of Computer Science, Purdue University, West Lafayette, IN, March 1992.

[11] G. N. FREDERICKSON, M. S. HECHT, AND C. E. KIM, *Approximation algorithms for some routing problems*, SIAM J. Comput., 7 (1978), pp. 178–193.

[12] H. N. GABOW, Z. GALIL, T. SPENCER, AND R. E. TARJAN, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica, 6 (1986), pp. 109–122.

[13] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 416–429.

[14] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[15] D. HAREL AND R. E. TARJAN, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[16] R. M. KARP, private communication, 1988.

[17] ———, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 85–103.

[18] D. E. KNUTH, private communication, 1988.

[19] ———, *The Art of Computer Programming. Volume* III: *Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[20] R. MCNAUGHTON, *Scheduling with deadlines and loss functions*, Management Sci., 6 (1959), pp. 1–12.

[21] B. SCHIEBER AND U. VISHKIN, *On finding lowest common ancestors: Simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.

[22] R. E. TARJAN, *Finding optimum branchings*, Networks, 7 (1977), pp. 25–35.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1153–1160, December 1992
© 1992 Society for Industrial and Applied Mathematics
009

LOWER BOUNDS ON UNIVERSAL TRAVERSAL SEQUENCES FOR CYCLES AND OTHER LOW DEGREE GRAPHS*

MARTIN TOMPA†

Abstract. Universal traversal sequences for cycles require length $\Omega(n^{1.29})$, improving the previous bound of $\Omega(n \log n)$. For $d \geq 3$, universal traversal sequences for d-regular graphs require length $\Omega(d^{0.71} n^{2.29})$. For constant d, the best previous bound was $\Omega(n^2 \log n)$.

Key words. graph traversal, cycle, lower bound, universal traversal sequence, circumnavigation, reflecting sequence

AMS(MOS) subject classifications. 68Q25, 68R10

1. Reflecting and universal traversal sequences. Universal traversal sequences were introduced by Cook (see Aleliunas et al. [2]) as a particularly simple method for traversing graphs. Universal traversal sequences are defined as follows.

For $2 \leq d < n$, let $\mathcal{G}(d,n)$ be the set of all connected, d-regular, n-vertex, edge-labeled, undirected graphs $G = (V, E)$. For this definition, edges are labeled as follows. For every edge $\{u,v\} \in E$ there are two labels $l_{u,v}$ and $l_{v,u}$ with the property that, for every $u \in V$, $\{l_{u,v} \mid \{u,v\} \in E\} = \{0, 1, \cdots, d-1\}$. For such labeled graphs, a string over $\{0, 1, \cdots, d-1\}$ can be thought of as a sequence of edge traversal commands. In particular, any $U = U_1 U_2 \cdots U_k \in \{0, 1, \cdots, d-1\}^*$ and $v_0 \in V$ determine a unique sequence $(v_0, v_1, \cdots, v_k) \in V^{k+1}$ such that $l_{v_{i-1},v_i} = U_i$ for all $i \in \{1, 2, \cdots, k\}$. Such a sequence U is said to *traverse* G starting at v_0 if and only if every vertex in G appears at least once in the sequence $v_0, v_1, \cdots, v_k$. U is a *universal traversal sequence* for $\mathcal{G}(d,n)$ if and only if U traverses each $G \in \mathcal{G}(d,n)$ starting at any vertex in G. $U(d,n)$ denotes the length of a shortest universal traversal sequence for $\mathcal{G}(d,n)$. To avoid trivialities, define $U(d,n) = U(d, n+1)$ in case $\mathcal{G}(d,n)$ is empty, which occurs exactly when both d and n are odd (see [5, Prop. 1]).

Good bounds on $U(d,n)$ translate into good bounds on the time required by certain simple undirected graph traversal algorithms running in very limited space. In particular, determining good lower bounds on $U(d,n)$ is a prerequisite to proving time-space tradeoffs for traversing undirected graphs. (See Borodin, Ruzzo, and Tompa [5] for a more detailed discussion.) Proving such lower bounds is the emphasis of this paper. The current best upper and lower bounds on $U(d,n)$ are summarized in Table 1. (See Borodin, Ruzzo, and Tompa [5] for more background.) Prior to the current work, the best lower bound for $d = 2$ was

$$U(2,n) = \Omega(n \log n),$$

due to Bar-Noy et al. [4]. This bound is improved in Corollary 13 to

$$U(2,n) = \Omega(n^{\log_4 6}) = \Omega(n^{1.29}).$$

For $3 \leq d \leq n/3 - 2$, the best previous lower bound was

$$U(d,n) = \Omega\left(dn^2 \log \frac{n}{d} + d^2 n^2\right),$$

*Received by the editors November 12, 1990; accepted for publication (in revised form) September 24, 1991. This material is based upon work supported in part by IBM research contract 16980043.

†Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195.

TABLE 1
Best known bounds on length of universal traversal sequences.

Bound	Relevant Range	Source
$U(d,n) = O(n^3)$	$d = 2$	Aleliunas [1]
$U(d,n) = O(dn^3 \log n)$	$3 \le d \le n/2 - 1$	Kahn et al. [7]
$U(d,n) = O(n^3 \log n)$	$n/2 - 1 < d$	Chandra et al. [6]
$U(d,n) = \Omega(n^{1.29})$	$d = 2$	this paper
$U(d,n) = \Omega(d^{0.71} n^{2.29})$	$3 \le d < n^{\log_6 1.5}$	this paper
$U(d,n) = \Omega(d^2 n^2)$	$n^{\log_6 1.5} \le d \le n/3 - 2$	Borodin et al. [5]
$U(d,n) = \Omega(n^2)$	$n/3 - 2 < d$	Alon et al. [3]

due to Borodin, Ruzzo, and Tompa [5]. This bound is improved in Corollary 14 to

$$U(d,n) = \Omega(d^{2-\log_4 6} n^{1+\log_4 6} + d^2 n^2) = \Omega(d^{0.71} n^{2.29} + d^2 n^2).$$

The value of d at which the second term begins to dominate is $d = n^{\log_6 1.5} \approx n^{0.23}$.

By a clever extension of the techniques in this paper, Coppersmith [personal communication] has improved these bounds to $U(2,n) = \Omega(n^{1.33})$ and $U(d,n) = \Omega(d^{0.67} n^{2.33})$ for $d \ge 3$.

The method underlying the new lower bounds depends on analyzing traversals of "labeled chains." A *labeled chain of length* n is an undirected graph $G = (V, E)$, where $V = \{0, 1, \cdots, n\}$ and $E = \{\{i, i+1\} \mid 0 \le i < n\}$, with edge labels as follows. Every edge $\{i, i+1\}$ has two labels, $l_{i,i+1}$ and $l_{i+1,i}$, each a nonempty subset of $\{0, 1\}$, restricted as follows:

1. $l_{0,1} = l_{n,n-1} = \{0, 1\}$, and
2. One of $l_{i,i-1}$ and $l_{i,i+1}$ is $\{0\}$, and the other is $\{1\}$, for all $0 < i < n$.

Let $\mathcal{L}(n)$ be the set of labeled chains of length n. Thus, $|\mathcal{L}(n)| = 2^{n-1}$. A labeled chain G will often be identified with the string $\beta = \beta_1 \beta_2 \cdots \beta_{n-1} \in \{0,1\}^{n-1}$, where $l_{i,i+1} = \{\beta_i\}$ for $0 < i < n$. The string β is called the *label* of G.

Given a labeled chain G of length n, a string $U \in \{0,1\}^*$ can be considered as a sequence of edge traversal commands starting at vertex 0. If $U = U_1 U_2 \cdots U_{|U|}$, where $U_i \in \{0, 1\}$ for $1 \le i \le |U|$, U determines a unique sequence $(v_0 = 0, v_1, v_2, \cdots, v_{|U|}) \in \{0, 1, \cdots, n\}^{|U|+1}$ such that $U_i \in l_{v_{i-1}, v_i}$ for all $1 \le i \le |U|$. Such a sequence U is said to *reflect t times* on G if and only if there exist $0 < j_1 < j_2 < \cdots < j_t \le |U|$ such that $v_{j_{2k-1}} = n$ for all $1 \le k \le \lceil t/2 \rceil$ and $v_{j_{2k}} = 0$ for all $1 \le k \le \lfloor t/2 \rfloor$. U is a *t-reflecting sequence* for $\mathcal{L}(n)$ if and only if U reflects t times on each $G \in \mathcal{L}(n)$. $R(t,n)$ denotes the length of a shortest t-reflecting sequence for $\mathcal{L}(n)$.

The remainder of this section demonstrates how lower bounds on $R(t,n)$ imply lower bounds on the lengths of universal traversal sequences. Sections 2 and 3 establish the lower bound $R(t,n) \ge t n^{\log_4 6} > t n^{1.29}$. When combined with the reductions of Theorem 1 and Corollary 4 below, this yields lower bounds on the length of universal traversal sequences.

For $\beta \in \{0,1\}^*$, define $\bar{\beta}$ to be the string that results from reversing β and then complementing its bits. For instance, if $\beta = 00010$, then $\bar{\beta} = 10111$.

The first reduction shows how a lower bound on $R(1,n)$ implies a lower bound on the length of universal traversal sequences on cycles.

THEOREM 1. *For any positive integer n,* $U(2,2n) \geq R(1,n)$.

Proof. Let U be a universal traversal sequence for $\mathcal{G}(2,2n)$. It will be shown that U is a 1-reflecting sequence for $\mathcal{L}(n)$. Choose any labeled chain $G_\beta \in \mathcal{L}(n)$ with label $\beta \in \{0,1\}^{n-1}$. Construct a $2n$-cycle C_β whose clockwise label from its start vertex is $0\beta 0\bar{\beta}$. By induction on $|U|$, it is straightforward to show that the traversal of C_β according to U terminates at a vertex of distance j from the start vertex if and only if the traversal of G_β according to U terminates at vertex j. Since U is universal for $\mathcal{G}(2,2n)$, the vertex at distance n from the start vertex is reached during the traversal of C_β according to U. Hence, U reflects at least one time on G_β. □

Corollary 4 below demonstrates that a lower bound on the length of general reflecting sequences implies a lower bound on $U(d,n)$ for $d \geq 3$ as well. The key to relating reflecting sequences to universal traversal sequences of degree $d \geq 3$ lies in the notion of "circumnavigation sequences," introduced by Borodin, Ruzzo, and Tompa [5]. For any labeled cycle $C \in \mathcal{G}(2,n)$, a string over $\{0,1\}$ can be interpreted as a traversal sequence. In particular, any $U \in \{0,1\}^*$ and start vertex v_0 of C determine a unique sequence $(v_0, v_1, \cdots, v_{|U|})$ of vertices traversed by U. Such a sequence U is said to circumnavigate C t times starting at v_0 if there are at least t times at which the sequence returns to v_0 moving in the same direction in which it last left v_0. More precisely, *U circumnavigates C t times* if and only if there exist $0 \leq i_1 < i_2 \leq i_3 < i_4 \leq \cdots \leq i_{2t-1} < i_{2t} \leq |U|$ such that

1. $v_0 = v_{i_1} = v_{i_2} = \cdots = v_{i_{2t}}$;
2. $v_l \neq v_0$ for all $i_{2j-1} < l < i_{2j}$ and $1 \leq j \leq t$; and
3. $v_{i_{2j-1}+1} \neq v_{i_{2j}-1}$ for all $1 \leq j \leq t$.

U is a *t-circumnavigation sequence* for $\mathcal{G}(2,n)$ if and only if U circumnavigates each $C \in \mathcal{G}(2,n)$ t times starting at any vertex in C. $C(t,n)$ denotes the length of a shortest t-circumnavigation sequence for $\mathcal{G}(2,n)$.

The next theorem, due to Borodin, Ruzzo, and Tompa [5], relates universal traversal sequences of higher degree graphs to circumnavigations of cycles.

THEOREM 2 [5, §4.1]. *Let $d \geq 3$ be an integer and n be a multiple of* $8(d-1)$. *Then*

$$U(d,n) \geq \frac{d}{2} C\left(\frac{(d-2)n}{4} + 2, \frac{n}{8(d-1)}\right).$$

Using the same reduction as in the proof of Theorem 1, it is straightforward to relate reflecting sequences to circumnavigation sequences.

THEOREM 3. *For any positive integers t and n,* $C(t,2n) \geq R(2t,n)$.

Combining Theorems 2 and 3 gives the desired reduction.

COROLLARY 4. *Let $d \geq 3$ be an integer and n be a multiple of* $16(d-1)$. *Then*

$$U(d,n) \geq \frac{d}{2} R\left(\frac{(d-2)n}{2} + 4, \frac{n}{16(d-1)}\right).$$

2. A lower bound on $R(t,n)$.

THEOREM 5. *For any positive integers t, m, and n,*

$$R(t,mn) \geq R(R(t,m),n).$$

Proof. Let $U = U_1 U_2 \cdots U_{|U|}$ be a t-reflecting sequence for $\mathcal{L}(mn)$. It will be shown that U is an $R(t,m)$-reflecting sequence for $\mathcal{L}(n)$. Suppose this is not the case. Then

there is a chain $G_\beta \in \mathcal{L}(n)$ with label $\beta \in \{0,1\}^{n-1}$ such that U reflects s times on G_β, but not $s+1$ times, where $s < R(t,m)$.

Consider a traversal of G_β according to U, and let $v_0 = 0, v_1, \cdots, v_{|U|}$ be the sequence of vertices visited during this traversal. The traversal will be said to be at vertex *v at time i* if and only if $v_i = v$. Let $j_0 = 0 \le i_1 < j_1 \le i_2 < j_2 \le \cdots \le i_s < j_s \le |U|$ satisfy the following:

1. For all $1 \le k \le \lceil s/2 \rceil$, i_{2k-1} is the last time the traversal is at vertex 0 before it next reaches vertex n, which occurs at time j_{2k-1}, and

2. For all $1 \le k \le \lfloor s/2 \rfloor$, i_{2k} is the last time the traversal is at vertex n before it next reaches vertex 0, which occurs at time j_{2k}.

Let $U' = U_{i_1+1}U_{i_2+1}\cdots U_{i_s+1}$. Since $s < R(t,m)$, U' is not a t-reflecting sequence for $\mathcal{L}(m)$. That is, there is a chain $G_\alpha \in \mathcal{L}(m)$ with label $\alpha = \alpha_1\alpha_2\cdots\alpha_{m-1} \in \{0,1\}^{m-1}$ such that U' does not reflect t times on G_α.

Now construct $G \in \mathcal{L}(mn)$ with label

$$\beta\alpha_1\bar{\beta}\alpha_2\beta\alpha_3\bar{\beta}\alpha_4\cdots\alpha_{m-1}\tilde{\beta},$$

where $\tilde{\beta} = \beta$ if m is odd and $\tilde{\beta} = \bar{\beta}$ if m is even. The contradiction will result because G is constructed so that U "behaves the same" on G as U' does on G_α, yet U reflects t times on G (since U is a t-reflecting sequence for $\mathcal{L}(mn)$), but U' does not on G_α.

Example. Before continuing the proof, the construction will be illustrated by showing that

$$U = 111001000100111001000110110001000110111001000100110001000100$$

is not a 2-reflecting sequence for $\mathcal{L}(12)$. In this example, $m = 3$, $n = 4$, and $t = 2$. For the first step, U is not a 9-reflecting sequence for $\mathcal{L}(4)$. For instance, U reflects only 8 times on the chain with label $\beta = 011$. For this choice of β, the subsequence U' at which each of the 8 reflections begins is underlined below:

$$U = 1110010001\underline{0}011\underline{1}0010001\underline{1}011\underline{0}0010001\underline{1}011\underline{1}0010001\underline{0}011\underline{0}001000100.$$

That is, $U' = 01101100$. Now U' is not a 2-reflecting sequence for $\mathcal{L}(3)$. For instance, U' reflects only one time on the chain with label $\alpha = 01$. Hence, U itself reflects only one time on the chain with label

$$\overbrace{011}^{\beta}\,0\,\overbrace{001}^{\bar{\beta}}\,1\,\overbrace{011}^{\beta}.$$

Now that the example is complete, the proof of Theorem 5 will be continued. The phrase "behaves the same" more precisely means that the traversal of G according to U is at vertex hn at time j_k if and only if the traversal of G_α according to U' is at vertex h at time k for all $0 \le k \le s$. This will be proved by induction on k. Once it has been established, it implies that the number of reflections of U on G and U' on G_α is the same (since no additional reflections on G can be completed between times j_s and $|U|$), which completes the contradiction.

Basis ($k = 0$). At time $j_0 = 0$ of the traversal according to U on G, the traversal is at vertex $0 = 0 \cdot n$. At time $k = 0$ of the traversal according to U' on G_α, the traversal is also at vertex 0.

Induction ($k > 0$). Suppose that the traversal of G_α according to U' is at vertex h at time $k-1$, and the traversal of G according to U is at vertex hn at time j_{k-1}. Assume that

h is even, the case when h is odd being dual. Then k is odd, since on G_α the traversal is at an even numbered vertex at time $k-1$. Now between times j_{k-1} and i_k of a traversal according to U on G_β, the traversal starts and ends at vertex 0 without reaching vertex n. Therefore, between times j_{k-1} and i_k of the traversal according to U on G, the traversal starts and ends at vertex hn without reaching either vertex $(h-1)n$ or vertex $(h+1)n$. This is because the only bit from α that affects the traversal during this time interval is α_h, the label at vertex hn, and the traversal is independent even of α_h, except for reflection about vertex hn.

At this point, consider the next step of U' on G_α in conjunction with the next step of U on G. In both cases the label at the current vertex v is $l_{v,v+1} = \{\alpha_h\}$ (unless $h \in \{0, m\}$), and the next bit of the reflecting sequence is U_{i_k+1}. Thus, both traversals move in the same direction in the next step (even if $h \in \{0, m\}$). Assume without loss of generality that both move to the next greater vertex $v+1$. It suffices to prove, then, that the traversal of G according to U is at vertex $(h+1)n$ at time j_k.

Now between times i_k and j_k of a traversal according to U on G_β, the traversal starts at vertex 0 and ends at vertex n, without repeating either. Therefore, between times i_k and j_k of the traversal according to U on G, the traversal starts at vertex hn and ends at vertex $(h+1)n$. This is because, by assumption, its first step is to vertex $hn+1$, after which no bit from α affects it during this time interval. □

COROLLARY 6. *Let c and r be positive integers such that, for all positive integers t, $R(t,c) \geq rt$. Then if t is any positive integer and $n = c^m$ for some nonnegative integer m,*

$$R(t,n) \geq tn^{\log_c r}.$$

Proof. It is equivalent to prove that $R(t, c^m) \geq tr^m$, which will be done by induction on m.

Basis ($m = 0$). $R(t, 1) = t$.

Induction ($m > 0$). Assume by the induction hypothesis that $R(t, c^{m-1}) \geq tr^{m-1}$. By Theorem 5,

$$\begin{aligned} R(t, c^m) &\geq R(R(t, c^{m-1}), c) \\ &\geq rR(t, c^{m-1}) \\ &\geq tr^m. \end{aligned}$$
□

COROLLARY 7. *Let c and r be positive integers such that, for all positive integers t, $R(t,c) \geq rt$. Then $U(2,n) = \Omega(n^{\log_c r})$.*

Proof. From Theorem 1 and Corollary 6, $U(2, 2n) \geq n^{\log_c r}$ whenever n is an integral power of c. The Ω bound follows since $U(2, n)$ is monotone nondecreasing in n. □

COROLLARY 8. *Let c and r be positive integers such that, for all positive integers t, $R(t,c) \geq rt$. Then for any integer $3 \leq d \leq n/17 + 1$,*

$$U(d,n) = \Omega(d^{2-\log_c r} n^{1+\log_c r}).$$

Proof. From Corollaries 4 and 6,

$$U(d, n') \geq \frac{1}{4} d(d-2)n' \left(\frac{n'}{16(d-1)}\right)^{\log_c r}$$

whenever n' is a multiple of $16(d-1)$ and $n'/16(d-1)$ is an integral power of c. Let m be the integer satisfying

$$16(d-1)c^m \leq n - (d-1) < 16(d-1)c^{m+1}.$$

Let $n' = 16(d-1)c^m$, so that $(n-(d-1))/c < n' \le n-(d-1)$. Although $U(d,n)$ is not known to be monotone in n, it is true that $U(d,n) \ge U(d,n')$ (Borodin, Ruzzo, and Tompa [5, §3]). Hence,

$$\begin{aligned} U(d,n) &\ge U(d,n') \\ &\ge \frac{1}{4}d(d-2)n'\left(\frac{n'}{16(d-1)}\right)^{\log_c r} \\ &> \frac{1}{4}d\left(\frac{d}{3}\right)\frac{n-(d-1)}{c}\left(\frac{n-(d-1)}{16cd}\right)^{\log_c r} \\ &\ge \frac{1}{12}d^2\frac{16n}{17c}\left(\frac{n}{17cd}\right)^{\log_c r} \\ &= \Omega\left(d^2 n\left(\frac{n}{d}\right)^{\log_c r}\right). \end{aligned}$$

□

3. The basis. Corollary 11 below demonstrates that $R(t,4) \ge 6t$, yielding the exponent $\log_4 6 \approx 1.29$. In order to present the techniques needed in a simpler setting, Theorem 9 demonstrates that $R(t,3) \ge 4t$, which yields the slightly weaker exponent $\log_3 4 \approx 1.26$. It is easy to demonstrate that $R(t,2) \le 2t+2$, so that no nontrivial lower bound can be derived using chains of length 2 as the basis.

THEOREM 9. *$R(2t-1,3) \ge 8t$ for all positive integers t.*

Proof. Let U be a $(2t-1)$-reflecting sequence for $\mathcal{L}(3)$ and $G_{00} \in \mathcal{L}(3)$ be the chain with label 00. For every traversal from vertex 0 to vertex 3 that U causes on G_{00}, consider the last time at which vertex 1 is left before vertex 3 is reached. U must have an occurrence of the (contiguous) substring 00 beginning at an even index to account for this event because the traversal is only at vertex 1 at odd times. There are at least t traversals from vertex 0 to vertex 3; mark any t of these even indices in U with the label "00." Using the analogous process, mark t even indices in U with each of "01," "10," and "11." Since no two of these marks can share the same even index, U must have at least $4t$ even indices; that is, $|U| \ge 8t$. □

The bound in Theorem 9 is tight: the string $1101(10001101)^t$ is a $(2t)$-reflecting sequence for $\mathcal{L}(3)$, as can be verified by applying it to the 4 labeled chains of length 3.

THEOREM 10. *$R(2t-1,4) \ge 12t$ for all positive integers t.*

Proof. Let U be a $(2t-1)$-reflecting sequence for $\mathcal{L}(4)$, and $G_{000} \in \mathcal{L}(4)$ be the chain with label 000. For every traversal from vertex 0 to vertex 4 that U causes on G_{000}, consider the last time at which vertex 1 is left before vertex 4 is reached. Although U need not have an occurrence of the (contiguous) substring 000, U must have an occurrence of the substring 00 beginning at an even index to account for this event. There are at least t traversals from vertex 0 to vertex 4; mark any t of these even indices in U with the label "000." Using the analogous process, mark t even indices in U with each of "001," "100," and "101." Despite the fact that two of these marks might share a single even index, it will be shown below that these $4t$ marks account for at least $3t$ distinct occurrences of the substrings 00 and 10 beginning at even indices. By an analogous argument, the other four marks "010," "011," "110," and "111" account for at least $3t$ distinct occurrences of the substrings 01 and 11 beginning at even indices. Since these two sets of indices must be disjoint, U has at least $6t$ even indices; that is, $|U| \ge 12t$.

Let the marks "000" and "101" count for two points each, and the marks "001" and "100" count for one point each, so that the $4t$ marks in U account for $6t$ points overall.

Either of the substrings 00 or 10 beginning at an even index will be called simply a *pair*. The remainder of this proof demonstrates that U cannot average more than two points per pair, so there must be at least $3t$ pairs. Notice that the substring of U that begins at an even index marked "000" must be of the form $00(10)^*0$. Table 2 lists the substrings corresponding to the other marks.

TABLE 2
Substrings of U beginning at marks.

Beginning at an even index with mark:	The characters of U must match:
000	$00(10)^*0$
001	$00(00)^*1$
100	$10(10)^*0$
101	$10(00)^*1$

Notice the following facts:

1. Any pair with two marks must be followed immediately by another occurrence of either 00 or 10; this is evident from Table 2;

2. Two pairs consecutive in U, either one marked "000" and the other marked "101," must be followed immediately by a third occurrence of either 00 or 10, again evident from Table 2;

3. Two pairs with the same mark must be separated by a distance of at least 8 to account for two reflections.

The first case to consider is two pairs consecutive in U, each with two marks. By fact (1), these must be followed immediately by a third occurrence of either 00 or 10. By fact (3), this third pair must be unmarked. Hence, in this case, 3 pairs accumulate 6 points.

The next case to consider is a pair with two marks, followed immediately by a pair with either of the single marks "000" or "101." By fact (2), these must be followed immediately by a third occurrence of either 00 or 10. If this third pair is marked at all, it must be the single mark "001" or "100," respectively, by fact (3). Hence, in this case again, 3 pairs accumulate at most 6 points.

Any other pair with two marks must be followed immediately either by an unmarked pair or by a pair with either of the single marks "001" or "100." In this case, 2 pairs accumulate at most 4 points.

This leaves only pairs with single marks, in which case 1 pair accumulates at most 2 points. □

The bound in Theorem 10 is tight: the string $10010001(101110010001)^t$ is a $(2t)$-reflecting sequence for $\mathcal{L}(4)$, as can be verified by applying it to the 8 labeled chains of length 4.

COROLLARY 11. *$R(t, 4) \geq 6t$ for all positive integers t.*

Proof.

Case 1. $t = 2k - 1$ for some integer k. Then

$$R(t, 4) = R(2k - 1, 4) \geq 12k = 6t + 6.$$

Case 2. $t = 2k$ for some integer k. Then

$$R(t, 4) = R(2k, 4) \geq R(2k - 1, 4) \geq 12k = 6t. \qquad \square$$

Combining Corollary 11 with Corollaries 6–8 yields the following lower bounds.

COROLLARY 12. *If n is any integral power of 4, and t is any positive integer,*

$$R(t,n) \geq tn^{\log_4 6} > tn^{1.29}.$$

COROLLARY 13. $U(2,n) = \Omega(n^{\log_4 6}) = \Omega(n^{1.29})$.

COROLLARY 14. *For any integer* $3 \leq d \leq n/17 + 1$,

$$U(d,n) = \Omega(d^{2-\log_4 6} n^{1+\log_4 6}) = \Omega(d^{0.71} n^{2.29}).$$

Using only the simple ideas in the proof of Theorem 9, it is not difficult to prove that $R(t,5) \geq 8t$, but $\log_5 8$ is slightly less than $\log_4 6$. Determining the rate of growth of $R(t,5)$ is an open question, and could well lead to improved lower bounds on the lengths of universal traversal sequences.

Acknowledgments. Once again, I am grateful to Larry Ruzzo for encouragement and enlightening discussions. Don Coppersmith simplified the proof of Theorem 10.

REFERENCES

[1] R. ALELIUNAS, *A simple graph traversing problem*, Master's thesis, Dept. of Computer Science, University of Toronto, Tech. Report No. 120, 1978.

[2] R. ALELIUNAS, R. M. KARP, R. J. LIPTON, L. LOVÁSZ, AND C. RACKOFF, *Random walks, universal traversal sequences, and the complexity of maze problems*, in Proceedings of the 20th Annual Symposium on Foundations of Computer Science, San Juan, Puerto Rico, Oct. 1979, IEEE, pp. 218–223.

[3] N. ALON, Y. AZAR, AND Y. RAVID, *Universal sequences for complete graphs*, Discrete Appl. Math., 27 (1990), pp. 25–28.

[4] A. BAR-NOY, A. BORODIN, M. KARCHMER, N. LINIAL, AND M. WERMAN, *Bounds on universal sequences*, SIAM J. Comput., 18 (1989), pp. 268–277.

[5] A. BORODIN, W. L. RUZZO, AND M. TOMPA, *Lower bounds on the length of universal traversal sequences*, in Proceedings of the 21st Annual ACM Symposium on Theory of Computing, Seattle, WA, May 1989, pp. 562–573; Journal of Comput. System Sci., 1992, to appear.

[6] A. K. CHANDRA, P. RAGHAVAN, W. L. RUZZO, R. SMOLENSKY, AND P. TIWARI, *The electrical resistance of a graph captures its commute and cover times*, in Proceedings of the 21st Annual ACM Symposium on Theory of Computing, Seattle, WA, May 1989, pp. 574–586.

[7] J. D. KAHN, N. LINIAL, N. NISAN, AND M. E. SAKS, *On the cover time of random walks on graphs*, J. Theoret. Probab., 2 (1989), pp. 121–128.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1161–1179, December 1992
© 1992 Society for Industrial and Applied Mathematics
010

APPROXIMATION AND SMALL-DEPTH FREGE PROOFS*

STEPHEN BELLANTONI†, TONIANN PITASSI†, AND ALASDAIR URQUHART‡

Abstract. Ajtai [*Proceedings of the 29th Annual IEEE Symposium on the Foundations of Computer Science*, White Plains, NY, 1988, pp. 346–355; preliminary version] recently proved that if for some fixed d, every formula in a Frege proof of the propositional pigeonhole principle PHP_n has depth at most d, then the proof size is not less than any polynomial in n. By introducing the notion of an "approximate proof" this paper demonstrates how to eliminate the nonstandard model theory, including the nonconstructive use of the compactness theorem, from Ajtai's lower bound. An approximate proof is one in which each inference is sound on a subset of the possible truth assignments—possibly a different subset for each inference. This paper also shows how to improve the lower bound, giving a specific superpolynomial function ($n^{\Omega(\log^{[d+1]} n)}$) bounding the proof size from below.

Key words. lower bounds, complexity of propositional proof systems, proof theory

1. Introduction. A Frege proof is a sequence of propositional formulas, each of which is either an axiom instance or follows from previous formulas by one of a fixed set of inference rules. The pigeonhole principle can be expressed by a class of propositional formulas, $\{PHP_n : n \in N\}$, where PHP_n asserts that there is no one-one mapping from a set D_0 of size $n+1$ to a set D_1 of size n.

Ajtai [Ajt] recently proved that if for some fixed d, every formula in a Frege proof of PHP_n has depth at most d, then the proof size is not less than any polynomial in n. His proof, while combinatorial in part, is proven for a nonstandard model of Peano arithmetic; the compactness theorem is then applied to obtain the result for standard values of n.

We demonstrate how to eliminate nonstandard model theory from Ajtai's lower bound by introducing the notion of an "approximate proof." An approximate proof is one in which each inference is sound on a subset of the possible truth assignments—possibly a different subset for each inference.

Our notion of approximation resembles that of Razborov [Raz], where functions are approximated by introducing small errors at each gate. However, instead of approximating just one formula, we are approximating each formula in a sequence of related formulas. The approximation made for each individual formula changes how the formulas relate to each other: instead of each formula being a sound conclusion from previous formulas, the inference is only an "approximately sound" inference. The use of approximation gives a more direct lower bound proof than was obtained using nonstandard model theory.

In this paper we also improve on Ajtai's result, giving a specific superpolynomial function that bounds the Frege proof size from below. The bound is $n^{\Omega(\log^{[d+1]} n)}$, where $\log^{[d+1]} n$ is $d+1$ iterations of log. Although the possibility of an exponential bound remains open, we give a reason why the proof method cannot be improved to yield an exponential bound.

We also demonstrate that if the Frege proof is of polynomial size, then its depth must be $\Omega(\log^* n)$. This improves the statement that can be inferred from Ajtai's result, namely, that polynomial size proofs must have nonconstant depth.

Constant-depth lower bounds for PHP_n are related to the power of the systems of bounded arithmetic, $I\Delta_0(f)$, and $S_2(f)$. In particular, a superpolynomial bound for

*Received by the editors April 16, 1991; accepted for publication (in revised form) November 15, 1991.
†Department of Computer Science, University of Toronto, Toronto, Canada M5S-1A4.
‡Department of Philosophy and Computer Science, University of Toronto, Toronto, Canada M5S-1A4.

PHP_n implies that $I\Delta_0(f)$ cannot prove the sentence asserting the pigeonhole principle for f, while an exponential lower bound imples that $S_2(f)$ cannot prove the pigeonhole principle for f. See Paris, Wilkie, and Woods [PWW]; Paris and Wilkie [PW]; and Ajtai [Ajt] for discussions of this question.

Lower bounds for propositional proof systems also bear on broader complexity issues. For example, the problem "NP =? co-NP" is equivalent to the following problem: "Is there a propositional proof system in which the correctness of a derivation can be checked in polynomial time, and which admits polynomial size proofs of all tautologies?" [CR].

Our lower bound is proved using a particular Frege system over the basis $\{\vee, \neg\}$, but it holds for any Frege system: by a theorem of Cook and Reckhow [CR], all Frege systems are polynomially equivalent; and by examining their theorem we find that the small depth of proofs is preserved in the polynomial length simulation.

The base case of our result is a generalization of an argument originally given by Haken [Ha] (and later abstracted by Urquhart [Urq]) showing that any resolution refutation of PHP_n must contain a large clause.

As in previous results [FSS], [Ajt2], [H], [Ajt], [Y], [Si] involving bounded depth formulas, we proceed by induction on the depth. Applying a random restriction at each depth, we can simplify the formulas enough to reduce the depth, without simplifying the problem too much. However, instead of obtaining a depth $d-1$ proof of the (restricted) pigeonhole principle, which is completely sound, we obtain a depth $d-1$ "approximate" proof of the (restricted) pigeonhole principle, which is only approximately sound. This approximation is introduced using a "pseudocomplement" similar to Ajtai's.

2. Overview and definitions.

2.1. Overview. We encode PHP_n using $(n+1)n$ propositional variables, $\{P_{ij} : i \in D_0 \;\&\; j \in D_1\}$, where D_0 and D_1 are disjoint sets such that $|D_0| = n+1$ and $|D_1| = n$. Intuitively, $P_{ij} = 1$ if and only if i is mapped to j. Since our proof system will be a refutation system, we are concerned with the statement $\neg\mathrm{PHP}_n$, which can be written as the conjunction of the following *pigeonhole clauses*:

$$\bigvee\{P_{ij} : j \in D_1\}, \qquad i \in D_0;$$
$$\bigvee\{\neg P_{ik}, \neg P_{jk}\}, \quad i \neq j, \quad i, j \in D_0, \quad k \in D_1.$$

In a refutation, one starts with the negated clauses $\neg\mathrm{PHP}_n$ as axioms and then derives $\bigvee\{\}$, i.e., False. More exact definitions of the formal system are given below.

We obtain the lower bound by induction on the depth, d, of the Frege refutation. Applying a random restriction to the refutation, we can simplify the bottom levels so that each occurrence of negation at depth 3 of each formula is replaced by the "pseudocomplement." This reduces the depth of each formula to $d-1$, but the resulting sequence of formulas may now only be an approximate refutation.

An approximate refutation is a Frege refutation where each inference is sound with respect to a large subset of all truth assignments. In contrast, an inference in a regular Frege refutation is sound with respect to all truth assignments. Note that our notion of an approximate proof is a local one: each inference can be sound with respect to a different subset of truth assignments, and there may be no single assignment that validates all the inferences. A "good" approximation for an inference can be obtained if every OR of small ANDs at the bottom levels can be "covered" by a small set after the restriction is applied. This covering set, which was also used by Ajtai, is analogous to the set of

variables remaining after restriction in [FSS]; it is dissimilar to the minterms of Håstad [H].

We repeat the restriction argument $d - 2$ times to obtain an approximate depth 2 Frege refutation of the pigeonhole principle, i.e., a refutation in which each formula is an OR of small ANDs. We then apply one more restriction to obtain a refutation in which each formula in the proof is an OR of small ANDs, covered by a small set. The existence of such a refutation contradicts the base case, which states that any good approximation to a Frege proof of the pigeonhole principle must contain a formula that has no small covering set.

2.2. Definitions.

The system H. The lower bound for the pigeonhole clauses will be proven for the Frege refutation system H, described in Fig. 1, for unbounded fan-in formulas. This system is a modification of the inference system in Shoenfield [Sh, p. 21]. The formulas of H are unordered rooted trees defined inductively by the following rules: if γ is a set of variables, then $\bigvee\{\bigwedge\gamma\}$ is a formula, if A is a formula then $\neg A$ is a formula, and if Γ is a finite set of formulas, then $\bigvee\Gamma$ is a formula. Thus the system allows $\wedge$ only at the bottom level, and in fact requires $\wedge$'s there. This syntactic requirement simplifies the exposition.

Excluded Middle Axiom: $A \overset{\circ}{\vee} \neg A$

Weakening Rule: $$\frac{A}{(A \overset{\circ}{\vee} B)}$$

Cut Rule: $$\frac{(A \overset{\circ}{\vee} B),\ (\neg A \overset{\circ}{\vee} C)}{(B \overset{\circ}{\vee} C)}$$

Merging Rule: $$\frac{\bigvee(\{\bigvee\Gamma\} \cup \Delta)}{\bigvee(\Gamma \cup \Delta)}$$

Unmerging Rule: $$\frac{\bigvee(\Gamma \cup \Delta)}{\bigvee(\{\bigvee\Gamma\} \cup \Delta)}$$

FIG. 1. *Rules of the system H.*

In the schemas of Fig. 1, A, B, and C represent formulas, and Γ and Δ are finite sets of formulas.

Note. $A \overset{\circ}{\vee} B$ is the formula $A \vee B$ with the ORs merged together. More formally, $A \overset{\circ}{\vee} B = \bigvee(\text{DISJUNCTS}(A) \cup \text{DISJUNCTS}(B))$; where $\text{DISJUNCTS}(X)$ is the set of disjuncts of X if X is a disjunction and $\text{DISJUNCTS}(X) = \{X\}$ otherwise.

The *size* of a formula is one plus the number of occurrences of $\vee$ and $\neg$ in the formula; the size of a Frege proof is the sum of the sizes of the formulas occurring as lines in the proof. Since each formula consists of ORs of ANDs in the bottom 2 levels, and the rest of the gates are ORs and NOTs, the depth of a formula is 2 plus the number of alternations of ORs and NOTs. The depth of a Frege proof is the maximum depth of the formulas in the proof.

If α is a propositional formula in the ordinary sense of, say, [Sh], then we can transform it into a formula α^H of the system H as follows: write it using the basis $\neg$ and $\vee$; then replace every propositional variable P_{ij} with $\bigvee\{\bigwedge\{P_{ij}\}\}$; then merge together any adjacent $\bigvee$'s created at heights 2 and 3. For example, the images of the negated PHP_n clauses are

$$\bigvee\{\bigwedge\{P_{ij}\} : j \in D_1\}, \qquad i \in D_0;$$
$$(\neg \bigvee\{\bigwedge\{P_{ik}\}\}) \vee (\neg \bigvee\{\bigwedge\{P_{jk}\}\}), \qquad i \neq j \in D_0; \quad k \in D_1.$$

Now, given a set of ordinary propositional formulas, say, $\{\alpha_i\}$, and given another ordinary formula β, we define an *H-proof of β from α over D* to be a sequence of formulas such that the final formula is β^H, and each formula is either α_i^H for some i or follows from zero or more preceding formulas using one of the axioms or rules of H (see Fig. 1). A *refutation of α over D* is an H-proof of $\bigvee\{\}$ from α^H, in which each formula has map-size 1. (Requiring the map size to be 1 does not decrease the generality of the system, using the transformation $\cdot^H$ described above.)

It is easy to see that the system H is implicationally complete, using, for example, the fact that the propositional fragment of Shoenfield's system is implicationally complete. If $\{\alpha\}_i \vdash \beta$ in Shoenfield's system, then we can obtain an H-proof of β from α by replacing every line γ of the Shoenfield proof with γ^H; additionally, we must insert appropriate combinations of the merging and unmerging rules in H. This translation preserves polynomial size and constant depth.

The system H is not suited to a direct proof of the lower bound. We will describe a modified version of H, H', that allows certain unsound inferences to be made. In spite of this unsoundness, we can retain control over the complexity by severely restricting the type of unsound inference that we permit. The new inference system will contain all of the rules of H plus additional rules that allow us to replace $\neg A$ by the pseudocomplement of A, when A is of a simple form. In order to describe the pseudocomplement, we need some definitions.

Maps; t-disjunctions; covering sets; one-one assignments. First recall that the *variables over $D = D_0 \cup D_1$* are $\{P_{ij} : i \in D_0, j \in D_1\}$. A *map over D* is defined to be a conjunction of the form $\bigwedge \Gamma$, where Γ is a set of variables over D such that distinct variables in Γ have distinct left subscripts and distinct right subscripts. Maps describe bijections between subsets of D_0 and subsets of D_1. The *size* of a map $\bigwedge \Gamma$ is $|\Gamma|$; if the size of a map is bounded by t, it is said to be a t-map. An OR of maps is called a *map disjunction*; if all the maps are of size at most t, then it is a *t-disjunction*.

For a map disjunction G, define $\min(G)$ to be the disjunction obtained by deleting every map from G that implies some other map in G. For example, $P_{11} = \min((P_{11}) \vee (P_{11} \wedge P_{34}))$. In other words, we remove the map C from G if there is some other map $B \subsetneq C$ in G. Of course, G and $\min(G)$ have the same truth value on all assignments.

A formula A is *covered* by a set $V \subseteq D_0 \cup D_1$ if every variable in A has either its left or right subscript in V; A is *k-coverable* if it is covered by some set V of size k. We write $\mathrm{Cover}(X)$ for the size of the smallest covering set of X.

A map or formula B is *properly* covered by V if it is covered by V, and every element of V covers some variable of B; that is, if V covers B and every vertex in V is hit by an edge of B. Although every minimum-size covering is proper, the converse is not true.

A truth assignment φ over D is any total assignment of $\{0,1\}$ to the variables over D. An assignment φ is *one-one on V* if $\{\langle i,j\rangle : \varphi(P_{ij}) = 1 \text{ and } (i \in V \vee j \in V)\}$ is a bijection (a map) properly covered by V.

Conflicting maps; pseudocomplements. Two maps $\bigwedge \Gamma$ and $\bigwedge \Delta$ are said to *conflict* if there are variables $P_{ij} \in \Gamma$ and $P_{kl} \in \Delta$ so that either $i = k$ and $j \neq l$, or $j = l$ and $i \neq k$. Notice that there is no map that conflicts with $\bigwedge\{\}$.

If A is a map disjunction such that $\min(A)$ is covered by V, then the *pseudocomplement*, $c(A, V, D)$, of A with respect to V on universe D is the following map disjunction:

$\bigvee\{B :$ B is a map over D properly covered by V, and B conflicts with all maps in $A\}$.

Notice $\min(A)$ doesn't have to be properly covered by V, just covered.

FACT. *If map B is properly covered by V and conflicts with a map D covered by V, then B and D conflict at some point inside V.*

Proof. Let $i \in D$ be a point at which B and D conflict; if $i \notin V$, then consider the case $i \in D_0$, and let $j \neq k$ be such that $P_{ij} \in B$ and $P_{ik} \in D$. Since D is covered by V yet $i \notin V$, it must be that $k \in V$. Since V properly covers B, there is i' such that $P_{i'k} \in B$. Now $i \neq i'$ since B is a map. □

The complement we have defined is not quite the same as the complement defined by Ajtai for two reasons. First, we require the conflicting maps to be properly covered where Ajtai just requires them to be covered; we need this change to make the distribution lemma hold (see below). Second, we do not require that A be covered by V, only that $\min(A)$ be covered by V. This simplifies the conversion lemma (below), and is a harmless change: a map conflicts with all the maps of $\min(A)$ if and only if it conflicts with all the maps of A.

Making these changes to Ajtai's pseudocomplement does not spoil its key property: $c(A, V, D)$ is equivalent to $\neg A$ with respect to all truth assignments over D which are one-one on V. More exactly, we have the following easy lemma.

LEMMA 2.1 (complement property). *If φ is a truth assignment that is one-one on V, then $\varphi(c(A, V, D)) = \varphi(\neg A)$.*

Proof. If $V = \emptyset$, then, since V covers $\min(A)$, either $A = \bigvee\{\}$ or else $\bigwedge\{\}$ is a map in A; the lemma is easily seen to hold in these cases. Therefore, we assume $V \neq \emptyset$.

If $\varphi(c(A, V, D)) = 1$, then let B be a (nonempty) map in $c(A, V, D)$ set to 1 by φ. Every map C in A conflicts with B at some point in V; since φ is one-one on V and assigns 1 to all variables of B, φ assigns zero to the conflicting variable in C. Considering all C this implies $\varphi(A) = 0$, so $\varphi(\neg A) = 1$.

In the other direction, suppose $\varphi(A) = 0$. Let B be the (nonempty) map properly covered by V, induced by the assignment φ. That is , $B = \bigwedge\{P_{ij} : \varphi(P_{ij}) = 1$ and P_{ij} is covered by $V\}$. Considering any map $C \in A$, we have $\varphi(C) = 0$; therefore, there is a variable in C that is set to zero by φ (and covered by V). This variable conflicts with some variable in B because $\varphi(B) = 1$ and φ is one-one on V. We conclude $B \in c(A, V, D)$, and, therefore, $\varphi(c(A, V, D)) = 1$. □

The system H'; approximate refutations; t-soundness. The new proof system, H', is obtained by adding the following schemes to H for every map disjunction A and set V covering $\min(A)$:

Approximate Excluded Middle Axiom
$$A \mathring{\vee} c(A, V, D),$$

Approximate Cut Rule
$$\frac{(A \mathring{\vee} B), \ (c(A, V, D) \mathring{\vee} C)}{(B \mathring{\vee} C)}.$$

Notice that these inferences depend on the fixed set D in the same sense as the PHP clauses depend on D. However, V can vary.

It should also be noted that the approximate complements in different parts of an approximate proof can be defined relative to quite different sets.

Neither the approximate excluded middle axiom nor the approximate cut rule are logically sound; however, by the complement property they are sound for the class of assignments that define one-one maps on V.

More formally, an inference in an approximate proof is *t-sound* if there is a set $V \subseteq D_0 \cup D_1$ with $|V| \leq t$ so that any assignment that defines a one-one map on V and makes all premises of the inference true also makes the conclusion true. A sound rule of inference is a 0-sound rule. If $|V| = t$ is large, there are only a small number of truth assignments that are one-one on V, and hence the inference is not very sound. On the other hand, the smaller $|V| = t$ is, the closer the inference is to a perfectly sound inference.

LEMMA 2.2 (soundness fact). *The approximate rules are $|V|$-sound.*

Proof. This follows immediately from the complement property. □

Using this fact, we can slightly strengthen the notion of t-soundness as follows: a proof in H' is *strongly t-sound* if every inference is either 0-sound or is one of the approximate rules involving $c(A, V, D)$, where $|V| \leq t$. In other words, we strengthen the condition so we know that the particular set V used in taking the pseudocomplement is also the set that witnesses the t-soundness.

We can think of "strongly t-sound" as a *syntactic* condition that is used to guarantee that the *semantic* requirement, "t-sound," holds.

Restrictions; miscellany. In choosing random restrictions, we use the same probability space as Ajtai. Each random restriction defines a one-one function between a subset of D_0 and a subset of D_1. Specifically, the probability space $\Omega^{n,\epsilon}$ is the set of all pairs $\rho = \langle r, s\rangle$, where s is a subset of $D = D_0 \cup D_1$ such that $s_0 = s \cap D_0$ is uniformly chosen with size $n^\epsilon + 1$, and, separately, $s_1 = s \cap D_1$ is uniformly chosen with size n^ϵ; and r is a uniformly chosen bijection from $D_0 \setminus s$ to $D_1 \setminus s$.

Every $\rho = \langle r, s\rangle$ in $\Omega^{n,\epsilon}$ determines a unique *restriction*, ρ, of the variables P_{ij}, ($i \in D_0, j \notin D_1$) as follows.

$$\rho(P_{ij}) = \begin{cases} * & \text{if } i \in s_0 \text{ and } j \in s_1, \\ 1 & \text{if } i \notin s_0 \text{ and } j \notin s_1 \text{ and } r(i) = j, \\ 0 & \text{otherwise} . \end{cases}$$

Notice that variables assigned $*$ by ρ are variables on s.

We think of restrictions as being performed syntactically on a formula: to apply a restriction, we remove from each map those variables that the restriction sets to one; and we remove from a map disjunction those maps in which some variable was set to zero. Thus, for example, the identity $(A \overset{\circ}{\vee} B)\restriction_\rho = A\restriction_\rho \overset{\circ}{\vee} B\restriction_\rho$ holds. Of course, by the definition of a formula, a given map cannot appear twice in a disjunction. When we want to perform additional simplifications, we explicitly mention the $\min()$ operation.

The notation $\Pr^{n,\epsilon}_\rho[A]$ denotes the probability that A occurs when ρ is drawn from $\Omega^{n,\epsilon}$. For a Boolean formula F and an element $\rho \in \Omega_{n,\epsilon}$, F restricted by ρ will be denoted by $F\restriction_\rho$. The notation $\log^{[l]} n$ indicates l applications of the base-2 log function (*not* $(\log n)^l$).

Throughout this paper, D_0 is a set of size $n+1$ and D_1 is a set of size n; where it is convenient, we shall assume that an ordering is given for each of D_0 and D_1. Whenever we write a real number where an integer is required, we mean the integer part of the real number (floor). When we assert an inequality involving n, we shall often assume tacitly that n is sufficiently large.

3. Reducing the depth. In this section we show how a proof of depth d is converted into one of depth $d-1$, while preserving approximate soundness.

All formulas in the proof will be approximated simultaneously in a bottom-up, level-by-level fashion by repeatedly applying restrictions, replacing each negation at height 3 by an approximating OR, and merging, until we eventually obtain all depth 2 formulas. Note that while the approximation of each gate is quite good, an original depth d formula may eventually be transformed into a very different depth 2 formula. The key point is that our inference rules have the syntactic property that only one gate may be eliminated per inference, and hence our gate-by-gate approximation leads to a new sequence of formulas, which are still approximately sound.

At each stage, the depth is reduced by 1, and some of the inferences are converted from being 0-sound to being t'-sound for some t'. Inferences that were made t-sound in some previous stage will remain at worst t-sound; they will automatically be t'-sound since we will have $t' \geq t$.

In this connection, notice that in the cut rule (and in the approximate cut rule), replacing say B by an approximating formula B' in both the hypothesis and conclusion does not affect the soundness of the inference. The soundness of the inference is only affected when we approximate the negations at the top level of the formulas; for example, when we use the pseudocomplement on the negation that is explicitly mentioned in the cut rule.

We will need to prove that the conversion process results in a syntactically proper approximate proof; as a step towards this, we show in the following lemma that the pseudocomplement is in an appropriate sense invariant under restrictions.

LEMMA 3.1 (distribution lemma). *Let A be a map disjunction over D such that V covers* $\min(A)$; *and let $\rho = \langle r, s\rangle \in \Omega^{n,\epsilon}$. Then $c(A, V, D)\lceil_\rho = c(A\lceil_\rho, V\cap s, D\cap s)$.*

Proof. First we show that any given map B' in $c(A, V, D)\lceil_\rho$ is also in $c(A\lceil_\rho, V\cap s, D\cap s)$. Let B be a map in $c(A, V, D)$ such that $B\lceil_\rho = B' \neq 0$. We wish to show that $B\lceil_\rho$ is in $c(A\lceil_\rho, V\cap s, D\cap s)$. First of all, because B is properly covered by V and B contains only variables set to 1 or $*$ by ρ, $B\lceil_\rho$ is properly covered by $V\cap s$. Secondly, variables in $B\lceil_\rho$ are all variables over $D\cap s$ because other variables are assigned values by ρ; since $B\lceil_\rho$ is a map, it is a map over $D\cap s$. Now, if $A\lceil_\rho = 0 = \bigvee\{\}$, then trivially $B\lceil_\rho$ is in $c(A\lceil_\rho, V\cap s, D\cap s)$. Otherwise, let D' be any map in $A\lceil_\rho$, and let D be a map in A such that $D' = D\lceil_\rho$. By definition of $c(A, V, D)$, B conflicts with D; using symmetry, let us assume that P_{ij} is in B, and P_{ik} in D, where $j \neq k$. If P_{ij} (or P_{ik}) were set to zero by ρ, a contradiction would hold because then $B\lceil_\rho$ (or, respectively, $D\lceil_\rho$) would be zero and, therefore, would not be a map in $c(A, V, D)\lceil_\rho$ (respectively, $A\lceil_\rho$). And if either of the two variables were set to one, the other would be set to zero; hence $\rho(P_{ij}) = \rho(P_{ik}) = *$. Therefore, B' and D' conflict.

In the other direction, let $B' = \bigwedge\Gamma$ be any given map in $c(A\lceil_\rho, V\cap s, D\cap s)$. Define B by

$$\Delta = \{P_{ij} : \quad P_{ij} \text{ is a variable over } D, \quad P_{ij}\lceil_\rho = 1, \text{ and } P_{ij} \text{ covered by } V\},$$

$$B = \bigwedge(\Gamma \cup \Delta).$$

Notice that $(\bigwedge\Gamma)\restriction_\rho \neq 0$ because Γ consists of variables over $D \cap s$, and any variable set to zero by ρ is not a variable over $D \cap s$. Therefore, the variables in Γ and Δ don't conflict.

By construction, $B\restriction_\rho = B'$. We wish to show that B is in $c(A, V, D)$, implying $B' \in c(A, V, D)\restriction_\rho$. Firstly, B is a map over D covered by V, by construction. Secondly, V covers B properly because vertices of V are either in $V \setminus s$ and hit by edges of Δ, or are in $V \cap s$ and hit by edges of B', using the properness of $V \cap s$ for B'.

Now, if $A = 0 = \bigvee\{\}$, then trivially B conflicts with all maps in A and, therefore, is in $c(A, V, D)$; else consider any map C in A. We must show that B and C conflict. Notice that C is a map over D covered by V, and $C\restriction_\rho$ is a map over $D \cap s$ covered by $V \cap s$.

If $B\restriction_\rho$ conflicts with $C\restriction_\rho$, then B conflicts with C, as desired. Otherwise, $B\restriction_\rho$ doesn't conflict with $C\restriction_\rho$; since $B\restriction_\rho \in c(A\restriction_\rho, V \cap s, D \cap s)$, it must be that $C\restriction_\rho$ is not in $A\restriction_\rho$. Yet C is in A, and because $c(A\restriction_\rho, V \cap s, D \cap s) \neq 0$, $A\restriction_\rho \neq 1$. Since the restriction ρ is applied syntactically, it must be that ρ removes C from A, i.e., $C\restriction_\rho = 0$. This means some variable P_{xy} in C is set to zero.

By the properness of $V \cap s$ for Γ, if P_{xy} were covered by $V \cap s$, then it would either be in Γ (contradicting $B\restriction_\rho \neq 0$) or would conflict with Γ (implying that B conflicts with C, as desired). Since C is covered by V, the only remaining case is that P_{xy} is a variable covered by $V \setminus s$ and set to zero by ρ.

Let us say, using symmetry, that $y \in V \setminus s$. Now let P_{vy} be a variable set to 1 by ρ, and which, therefore, conflicts with P_{xy}. Since P_{vy} is in Δ, B conflicts with C. □

DEFINITION 3.2. An approximate refutation of PHP_n is (d, t)-*good* if it has depth at most d, map size at most t, and is strongly t-sound. Notice that if a refutation is (d, t)-good, then it is (d, t')-good for all $t' \geq t$.

Below, we will describe a sufficient condition that allows us to convert a (d, t)-good refutation into a $(d-1, t)$-good refutation. First we describe the conversion mechanism.

Let P be a (d, t)-good refutation over D of PHP_n $(d > 2)$, and let ρ be a restriction. P is converted into a depth $d - 1$ refutation in four steps.

(1) Let $G_0 \cdots G_m$ be the distinct map disjunctions appearing in formulas of $P\restriction_\rho$. (We only need consider *maximal* map disjunctions, which appear in $P\restriction_\rho$ as other than proper subformulas of map disjunctions.) Let $W_0 \cdots W_m \subseteq D \cap s$ be minimum size covering sets for $\min(G_0\restriction_\rho) \cdots \min(G_m\restriction_\rho)$. In case G is just $\bigvee\{\bigwedge\{P_{jk}\}\}$ for some j, k, then we prefer to cover $\min(G\restriction_\rho)$ with $W = \{k\}$.

(2) Apply the restriction ρ to each formula of P.

(3) Replace each occurrence of $\neg G_i\restriction_\rho$ by $c(G_i\restriction_\rho, W_i, D \cap s)$.

(4) Merge together OR gates appearing at heights 2 and 3 in the new proof.

LEMMA 3.3 (conversion lemma). *Let P be a (d, t)-good approximate refutation over D of* PHP_n $(d > 2)$, *and let* $\rho = \langle r, s\rangle \in \Omega^{n,\epsilon}$. *If* $t' \geq t$ *and* $Cover(\min(G\restriction_\rho)) \leq t'$ *for every maximal map disjunction G in P, then P converted by ρ is a* $(d-1, t')$*-good approximate refutation over $D \cap s$ of* $\mathrm{PHP}_{n^\epsilon}$.

Proof. Let $G_0 \cdots G_m$ and $W_0 \cdots W_m$ be as described above.

We must consider each inference of the original proof and see that after the conversion process, it remains a strongly sound inference in the system H'.

Suppose that the inference is $A \mathring{\vee} \neg A$ (excluded middle axiom). If $\mathrm{DEPTH}(A) > 2$, then the conversion results in another excluded middle axiom. This is strongly 0-sound and, therefore, strongly t'-sound. If $\mathrm{DEPTH}(A) = 2$ (i.e., $A = G_i$ is a map disjunction), then the conversion results in $G_i\restriction_\rho \mathring{\vee} c(G_i\restriction_\rho, W_i, D \cap s)$, an instance of the approximate excluded middle axiom. Since $|W_i| \leq t'$ is given, the instance is strongly t'-sound.

Suppose the inference is $(A \overset{\circ}{\vee} B)$, $(\neg A \overset{\circ}{\vee} C) \Rightarrow (B \overset{\circ}{\vee} C)$ (cut rule). If $\text{DEPTH}(A) > 2$, then the conversion results in another instance of the cut rule. If $\text{DEPTH}(A) = 2$, then $A = G_i$ for some i, and the conversion results in $(G_i \overset{\circ}{\vee} B){\restriction}_\rho$, $(c(G_i{\restriction}_\rho, W_i, D \cap s) \overset{\circ}{\vee} C{\restriction}_r$ ho) $\Rightarrow (B \overset{\circ}{\vee} C){\restriction}_\rho$; by the definition of $\restriction$ this is identically $(G_i{\restriction}_\rho \overset{\circ}{\vee} B{\restriction}_\rho,)$ $(c(G_i{\restriction}_\rho, W_i, D \cap s) \overset{\circ}{\vee} C{\restriction}_r$ ho) $\Rightarrow (B{\restriction}_\rho \overset{\circ}{\vee} C{\restriction}_\rho)$, a strongly $|W_i| \leq t'$ sound instance of the approximate cut rule over $D \cap s$.

If the inference is an instance of the weakening rule, the merging rule, or the unmerging rule, then the converted inference is an instance of the same rule. (Essentially, this holds because $\neg$ does not appear in these rules.)

Suppose the inference is $A \overset{\circ}{\vee} c(A, V, D)$ for some map disjunction A and some set V covering $\min(A)$ (approximate excluded middle axiom over D). The converted formula is $A{\restriction}_\rho \overset{\circ}{\vee} c(A, V, D){\restriction}_\rho$, which by the distribution lemma is $A{\restriction}_\rho \overset{\circ}{\vee} c(A{\restriction}_\rho, V \cap s, D \cap s)$, an instance of approximate excluded middle over $D \cap s$. Since $A \overset{\circ}{\vee} c(A, V, D)$ was a strongly t-sound instance, we have $|V| \leq t$, and, therefore, $|V \cap s| \leq t \leq t'$. Therefore, $A{\restriction}_\rho \overset{\circ}{\vee} c(A{\restriction}_\rho, V \cap s, D \cap s)$ is a strongly t'-sound inference.

Suppose the inference is $(A \overset{\circ}{\vee} B)$, $(c(A, V, D) \overset{\circ}{\vee} C) \Rightarrow (B \overset{\circ}{\vee} C)$ (approximate cut rule over D). Using the distribution lemma again, the converted inference is an instance of the approximate cut rule over $D \cap s$. Using reasoning similar to that for the approximate excluded middle axiom, the inference is strongly $t \leq t'$ sound.

Finally, we analyze the PHP_n clauses as follows.

- $\bigvee\{\bigwedge\{P_{ij}\} : j \in D_1\}$ becomes $\bigvee\{\bigwedge\{\}\}$ if $i \notin s$; this is an instance of the approximate excluded middle over $D \cap s$, with $A = \bigvee\{\}$ and $V = \emptyset$. If $i \in s$, it becomes a PHP_{n^ϵ} clause over $D \cap s$.
- For $(\neg \bigvee\{\bigwedge\{P_{ik}\}) \vee (\neg \bigvee\{\bigwedge\{P_{jk}\})$, recall that we preferred to cover both the disjuncts with $\{k\}$. Therefore, both complements will be taken with respect to $\{k\}$, and the result will be $\bigvee\{\bigwedge\{P_{hk}\} : h \in D_0 \cap s\}$. This is an instance of the approximate excluded middle over $D \cap s$, with $A = \bigvee\{\}$ and $V = \{k\}$.

The condition that maps in the converted proof be of size at most t' is easy because every map disjunction in the converted proof is t'-coverable. □

4. The lower bound. In this section the lower bound is stated and proven using the following lemma (see §5), which says that under suitable conditions, applying a random restriction to a map disjunction makes it coverable by a small set. (This lemma will be presented later in sequence.)

LEMMA 5.1 (covering lemma). *Let G be a t-disjunction, and let ϵ be a constant such that* $0 < \epsilon < 1/16$. *If* $t = o(\log \log n)$ *and* $8/\sqrt{\epsilon} \leq k \leq 2n^{\epsilon^{2t}}$, *then (for sufficiently large* n),

$$Pr_\rho^{n,\,\epsilon^{2t}}[Cover(\min(G{\restriction}_\rho)) > k] \leq \alpha_k^{n,t},$$

where

$$\alpha_k^{n,t} = \left(\frac{1}{n}\right)^{\frac{1}{11}\epsilon^{2t}k}.$$

It is convenient to introduce some constants and functions. We also indicate some relationships between the quantities.

$$\begin{aligned}
\epsilon &= \tfrac{1}{25},\\
c &= (2\log\tfrac{1}{\epsilon}) > 2,\\
t_d(n) &= \tfrac{1}{3c}\log^{[d+1]} n \le \delta_d(n) t_{d-1}(n^{\delta_d(n)}),\\
\delta_d(n) &= \epsilon^{2t_d(n)} = (\log^{[d]} n)^{-1/3},\\
S_d(n) &= n^{t_d(n)/11} \le S_{d-1}(n^{\delta_d(n)}).
\end{aligned}$$

The inequalities can be calculated as follows. First,

$$t_{d-1}(n^{\delta_d(n)}) = \frac{1}{3c}\log^{[d]}(n^{\delta_d(n)}) \ge \frac{1}{3c}\delta_d(n)\log^{[d]} n.$$

Since $\delta_d(n) = (\log^{[d]} n)^{-1/3}$, this gives $t_{d-1}(n^{\delta_d(n)}) \ge \frac{1}{3c}(\log^{[d]} n)^{2/3}$. Now since $2^{(c+1)t_d(n)} = (\log^{[d]} n)^{(c+1)/(3c)} < (\log^{[d]} n)^{1/2}$ (using $c > 2$), we obtain $t_{d-1}(n^{\delta_d(n)}) \ge 2^{(c+1)t_d(n)}$. Taking logs of both sides gives $\log t_{d-1}(n^{\delta_d(n)}) \ge ct_d(n) + t_d(n) \ge ct_d(n) + \log t_d(n)$. By the definitions above, this inequality is equivalent to all of the following, including the required inequalities:

$$\begin{aligned}
\log t_{d-1}(n^{\delta_d(n)}) &\ge \log t_d(n) - \log \delta_d(n),\\
\delta_d(n)\cdot t_{d-1}(n^{\delta_d(n)}) &\ge t_d(n),\\
(n^{\delta_d(n)})^{t_{d-1}(n^{\delta_d(n)})/11} &\ge (n)^{t_d(n)/11},\\
S_{d-1}(n^{\delta_d(n)}) &\ge S_d(n).
\end{aligned}$$

The lower bound will follow by induction on the depth using the following two lemmas, whose proofs will be given later.

LEMMA 4.3 (induction lemma). *For n sufficiently large, if $d > 2$ and P is any $(d, t_d(n))$-good refutation of* PHP_n *of size less than $S_d(n)$, then there is a restriction $\rho \in \Omega^{n,\delta_d(n)}$ such that: P converted by ρ is a $(d-1, t_{d-1}(n^{\delta_d(n)}))$-good refutation of* $\mathrm{PHP}(n^{\delta_d(n)})$ *of size less than $S_{d-1}(n^{\delta_d(n)})$.*

LEMMA 4.4 (base lemma). *For sufficiently large n, there is no $(2, t_2(n))$-good refutation of* PHP_n *of size less than $S_2(n)$.*

THEOREM 4.1 (lower bound on size). *For sufficiently large n, any refutation of* PHP_n *of depth d must have size at least $S_d(n) = n^{\Omega(\log^{[d+1]} n)}$.*

Proof. Suppose there were such a refutation, P. Since it is a refutation in H it has map size at most 1, and is strongly 0-sound; therefore, P is a $(d, t_d(n))$-good approximate refutation of PHP_n of size less than $S_d(n)$. Without loss of generality we can assume that P has depth at least 2; now applying the induction and base lemmas below gives a contradiction. □

THEOREM 4.2 (lower bound on depth). *For sufficiently large n, any Frege refutation of* PHP_n *of polynomial size must have depth $\Omega(\log^* n)$.*

Proof. The asymptotics in the lower bound on size hold at least for d up to $O(\log^* n)$. Supposing the size to be bounded by n^k and setting $n^k = S_d(n)$ gives $\log^{[d+1]} n = k$, i.e., $d = \Omega(\log^* n)$. □

To prove the following results, we let n be sufficiently large so that the required asymptotic relationships hold for all depths up to d, and then proceed by induction on the depth, d. Each time we reduce the depth by applying a restriction to a universe of size n, the size of the resulting universe is $n^{\delta_d(n)}$.

LEMMA 4.3 (induction lemma). *For n sufficiently large, if $d > 2$ and P is any $(d, t_d(n))$-good refutation of* PHP_n *of size less than $S_d(n)$, then there is a restriction $\rho \in \Omega^{n,\delta_d(n)}$ such that: P converted by ρ is a $(d-1, t_{d-1}(n^{\delta_d(n)}))$-good refutation of* $\mathrm{PHP}(n^{\delta_d(n)})$ *of size less than $S_{d-1}(n^{\delta_d(n)})$.*

Proof. For each map disjunction G in P, however, the covering lemma implies that $\mathrm{Cover}(\min(G\restriction_\rho)) > t_{d-1}(n^{\delta_d(n)})$ with probability at most $\alpha^{n,t_d(n)}_{t_{d-1}(n^{\delta_d(n)})} = 1/S_{d-1}(n^{\delta_d(n)})$. The conditions required by the covering lemma, that $t_d(n) = o(\log\log n)$ and that $t_{d-1}(n^{\delta_d(n)}) \leq n^{\delta_d(n)}$, are easily seen to hold for $d \geq 2$.

Since $1/S_{d-1}(n^{\delta_d(n)}) \leq 1/S_d(n)$, and there are fewer than $S_d(n)$ map disjunctions in P, the probability is less than one that some map disjunction has $\mathrm{Cover}(\min(G\restriction_\rho)) > t_{d-1}(n^{\delta_d(n)})$. In particular, there is a restriction ρ that makes all the map disjunctions coverable by small sets after taking $\min()$. Since $t_{d-1}(n^{\delta_d(n)}) \geq t_d(n)$, we can apply the conversion lemma to show that P converted by ρ is $(d-1, t_{d-1}(n^{\delta_d(n)}))$-good. The size of P after conversion is still at most $S_d(n)$, which is at most $S_{d-1}(n^{\delta_d(n)})$. □

It is interesting to notice that depth 2 proofs never contain any of the second type of PHP_n clause $(\neg\bigvee\bigwedge P_{ik} \vee \neg\bigvee\bigwedge P_{jk})$ because these have depth 3. These clauses all get converted into instances of approximate excluded middle the very first time a restriction is applied.

LEMMA 4.4 (base lemma). *For sufficiently large n, there is no $(2, t_2(n))$-good refutation of* PHP_n *of size less than $S_2(n)$.*

Proof. Suppose there were such a refutation, P. The same calculations as in the induction lemma allow us to use the covering lemma to show that there is a restriction $\rho \in \Omega^{n,\delta_2(n)}$ such that $\mathrm{Cover}(\min(G\restriction_\rho)) \leq t_1(n^{\delta_2(n)})$ for all maximal map disjunctions G in P. These maximal map disjunctions of P are exactly the formulas of P because $d = 2$.

Applying ρ to P and replacing each map disjunction X with $\min(X)$ gives $t_2(n)$-sound refutation P' of $\mathrm{PHP}_{n^{\delta_2(n)}}$ such that every formula of P' is $t_1(n^{\delta_2(n)})$-coverable. Since $t_1(n^{\delta_2(n)}) \leq n^{\delta_2(n)}/8$ for sufficiently large n, the existence of P' contradicts the criticality lemma below. We have not shown that P' is a refutation in H', but that doesn't matter to the criticality lemma. □

The criticality lemma, which provides the argument for the base case of the theorem, is a modification of Urquhart's argument [Urq] generalizing the resolution-system lower bound of Haken [Ha].

DEFINITION 4.5. An assignment is *i-critical* if it is one-one on $D_0\backslash\{i\}$ (and, therefore, is also one-one on D_1). For any formula A, the *critical* set $\mathrm{CRIT}(A)$ is defined by

$$\mathrm{CRIT}(A) = \{i : A\restriction_\rho = 0 \text{ for some } i\text{-critical } \rho\}.$$

LEMMA 4.6 (criticality lemma). *There is no $n/12$-sound approximate refutation of* PHP_n *in which all formulas are $(n/8-1)$-coverable.*

Proof. Suppose P were such an approximate refutation. Let A be the first formula in P such that $|\mathrm{CRIT}(A)| \geq n/3$. There is such a formula because $|\mathrm{CRIT}(\bigvee\emptyset)| = n+1$.

Let $\{B_1, \cdots B_k\}$ be the k preceding formulas from which A is derived, with $k \leq 2$ and $k \geq 0$. Since the inference is $n/12$-sound, there is a set V of size at most $n/12$ such that any assignment which is one-one on V and makes $B_1 \cdots B_k$ true, also makes A true. Whenever $i \notin V$ and φ is i-critical, φ is one-one on V, and, therefore, $A\restriction_\varphi = 0 \Rightarrow \exists l,\ B_l\restriction_\varphi = 0$. Hence

$$\mathrm{CRIT}(A) \subseteq V \cup \cup_{l=1}^{k} \mathrm{CRIT}(B_l).$$

Since $|\mathrm{CRIT}(B_l)| < n/3$ by the minimality of A, this implies $|\mathrm{CRIT}(A)| < n/12 + kn/3 \leq n/12 + 2n/3 = 3n/4$ and $|D_0 \setminus \mathrm{CRIT}(A)| \geq n/4$.

Now we use the facts that both $\textsc{crit}(A)$ and $D_0 \setminus \textsc{crit}(A)$ are large to show that A is not $(n/8-1)$-coverable. Specifically, we find $n/8$ variables in A that have disjoint subscripts.

For any i-critical assignment φ and $j \neq i$, let $r(j)$ be such that $\varphi(P_{j,r(j)}) = 1$. Now let $\varphi[j,i]$ be the assignment that agrees with φ, except that $\varphi[j,i](P_{j,r(j)}) = 0$ and $\varphi[j,i](P_{i,r(j)}) = 1$. By switching j and i in this way, we get $\varphi[j,i]$ to be j-critical; when $j \in D_0 \setminus \textsc{crit}(A)$, this implies that $A\restriction_{\varphi[j,i]} \neq 0$.

For each $i \in \textsc{crit}(A)$, fix an i-critical assignment φ such that $A\restriction_{\varphi} = 0$. Consider any $j \in D_0 \setminus \textsc{crit}(A)$. Since $A\restriction_{\varphi} = 0$, but $A\restriction_{\varphi[j,i]} \neq 0$, either $P_{j,r(j)}$ or $P_{i,r(j)}$ occurs in A. Let $\textsc{var}(\varphi, i)$ be the variables so discovered, among all $j \in D_0 \setminus \textsc{crit}(A)$. Since φ defines a one-one function and $|D_0 \setminus \textsc{crit}(A)| \geq n/4$, there are at least $n/4$ distinct variables in each $\textsc{var}(\varphi, i)$.

Case 1. For some i, $\textsc{var}(\varphi, i)$ contains at least $n/8$ variables of the form $P_{j,r(j)}$, $j \in D_0 \setminus \textsc{crit}(A)$. These variables have mutually disjoint subscripts because φ defines a one-one function (namely, r).

Case 2. Otherwise, for every $i \in \textsc{crit}(A)$, $\textsc{var}(\varphi, i)$ contains at least $n/8$ variables of the form $P_{i,r(j)}$ for some $j \in D_0 \setminus \textsc{crit}(A)$. There are at least $n/8$ values for i, and by considering each in turn we can select a matching of size $n/8$ from the variables in $\cup_i \textsc{var}(\varphi, i)$. $\square$

5. Covering lemma. In this section we prove the following covering lemma (Lemma 5.1), which states that if you apply a sufficiently strong restriction to a t-disjunction, then the result is probably k-coverable (for suitable t and k).

LEMMA 5.1 (covering lemma). *Let G be a t-disjunction, and let ϵ be a constant such that $0 < \epsilon < 1/16$. If $t = o(\log\log n)$ and $8/\epsilon \leq k \leq 2n^{\epsilon^{2t}}$ for sufficiently large n, then for sufficiently large n,*

$$Pr_{\rho}^{n,\epsilon^{2t}}[Cover(\min(G\restriction_{\rho})) > k] \leq \alpha_k^{n,t},$$

where

$$\alpha_k^{n,t} = \left(\frac{1}{n}\right)^{\frac{1}{11}\epsilon^{2t}k}.$$

The proof is a simplification of Ajtai's T2 [Ajt], in which we extract specific bounds from the combinatorics. The covering lemma demonstrates that with high probability, applying a random restriction to a map disjunction results in a formula that can be covered by a small set. It is proved using a combinatorial lemma (5.2), which we derived from Ajtai's lemma C1.[1] First we state Lemma 5.2, then go ahead with the proof of the covering lemma.

LEMMA 5.2. *Let $0 < \delta < 1/3$, $0 \leq \epsilon \leq \delta^2/4$ and let g be a function defined on $D_0 \cup D_1$, such that $g(x) \subseteq D_0 \cup D_1$, $|g(x)| \leq n^{1-\delta}$, and $x \notin g(x)$ for all $x \in D_0 \cup D_1$. Then for all $t > 4/\sqrt{\epsilon}$ and for all sufficiently large n we have*

$$Pr_{\langle r,s\rangle}^{n,\epsilon}\left[\left|\bigcup_{x\in s} g(x) \cap s\right| > t\right] \leq \beta_t^n$$

[1]Ajtai's lemma C1, appearing in [Ajt] and [Ajt3], contains an error in the statement of (**) and a consequent error in the application of (**). He shows the proof of the corrected (**) in a private communication, which does not comment on the application of (**).

where

$$\beta_t^n = n^{-t(\frac{\epsilon}{5})}.$$

Proof of Lemma 5.1. Given a fixed but sufficiently large value for n, the proof proceeds by induction on t.

Base case. For the base case ($t = 1$), write G in the form $G = \bigvee_{i \in D_0} \bigvee_{j \in W_i} P_{ij}$ for appropriate sets $W_i \subseteq D_1$. Let $B = \{i \in D_0 : |W_i| \geq n^{1-2\epsilon}\}$.

Taking cases on the size of B, suppose $|B| \geq n^{3\epsilon}$, so that for n sufficiently large, $|B \setminus s| \geq n^{5\epsilon/2}$. A restriction $\rho = \langle r, s\rangle$ can be chosen as follows: first choose $s = s_0 \cup s_1$, where $|s_1| = n^{\epsilon^2}$ and $|s_0| = |s_1| + 1$; let $B_s \subseteq B \setminus s$ be any particular subset of size $n^{5\epsilon/2}$; next, for each $i \in B_s$ in increasing order, choose $r(i)$ uniformly from the remaining elements of D_1; then choose the rest of r. Each time $r(i)$ is chosen for $i \in B_s$, the probability is $|(W_i \setminus s) \setminus \{r(k) : k \in B_s \text{ and } k < i\}|$ out of $|(D_1 \setminus s) \setminus \{r(k) : k \in B_s \,\&\, k < i\}|$ that $r(i) \in W_i$. Hence, for $i \in B_s$, the probability that $(\bigvee_{j \in W_i} P_{ij})\lceil_\rho = 1$ is at least $(n^{1-2\epsilon} - n^{\epsilon^2} - n^{5\epsilon/2})/(n - n^{\epsilon^2}) \geq 1/(2n^{2\epsilon})$. It follows that the probability of this happening for at least one of the $n^{5\epsilon/2}$ possible $i \in B_s$ is at least $1 - (1 - \frac{1}{2n^{2\epsilon}})^{n^{5\epsilon/2}}$. Since $(1 - \frac{1}{x})^x \leq \frac{1}{e}$, this probability is at least $1 - (\frac{1}{e})^{\frac{1}{2}n^{\epsilon/2}}$. Because $t = 1$ and $k \leq n^{\epsilon^2}$, this probability is greater than or equal to $1 - \alpha_k^{n,1}$ for n sufficiently large. Finally, whenever this happens (i.e., whenever $(\bigvee_{j \in W_i} P_{ij})\lceil_\rho = 1$ for any $i \in B \setminus s$), $G\lceil_\rho = 1$, and therefore $\text{Cover}(\min(G\lceil_\rho)) = 0$.

On the other hand, suppose that $|B| < n^{3\epsilon}$ and a random $\rho = \langle r, s\rangle$ is chosen from $\Omega^{n,\epsilon^{2t}}$.

Firstly, we show that with high probability $|B \cap s_0| \leq k/2$. Applying Lemma 5.3 below, with the parameters $A' = \{B\}$, $c' = 0$, $t' = k/2$, $\delta' = (1 - 3\epsilon)$, and $\epsilon' = \epsilon^2$, we obtain that the probability of $|B \cap s| > k/2$ is at most $2n^{-(1-3\epsilon-\epsilon^2)k/4}$.

Secondly, we show that with high probability all variables $\{P_{ij} \mid i \notin B\}$ in $G\lceil_\rho$ are covered by a subset of s_1 of size at most $k/2$. We apply Lemma 5.2 to the system

$$\{i \to W_i : i \in D_0 \setminus B\} \cup \{i \to \emptyset : i \in B\},$$

with parameters $\epsilon' = \epsilon^2$, $t' = k/2 > 4/\sqrt{\epsilon'}$, and $\delta' = \sqrt{4\epsilon'}$. The condition $x \notin g(x)$ required by the lemma is trivially satisfied. Lemma 5.2 implies that with probability at most $n^{-\epsilon^2 k/10}$, the subsystem fails to be $k/2$-coverable after a restriction from $\Omega^{n,\epsilon^{2t}}$ is chosen and applied.

The variables remaining in G after $\langle r, s\rangle$ is applied are those in the subsystem just described, plus some variables that are covered by $B \cap s$. (Variables in $\{P_{ij} : i \in B \setminus s,\ j \in W_i\}$ are all set to either zero or one.) It follows that with probability at most $2n^{-(1-3\epsilon-\epsilon^2)k/4} + n^{-\epsilon^2 k/10} \leq \alpha_k^{n,1}$, the function $\min(G\lceil_\rho)$ is not k-coverable.

Induction step. Let $G = G_1 \overset{\circ}{\vee} G_2$ be the t-disjunction that we wish to cover, where G_1 are those maps of size exactly one and G_2 are those maps of size at least two. We will obtain bounds on the probability that $\min(G_1\lceil_\rho)$ is not $k' = k/2$ covered and on the probability that $\min(G_2\lceil_\rho)$ is not $k' = k/2$ covered; adding these two bounds we will obtained the desired bound on the probability that $\min(G\lceil_\rho)$ is not k covered. Applying the inductive hypothesis to G_1 for $t = 1$ we get that the probability that $\min(G_1\lceil_\rho)$ is not k'-coverable is at most $\left(\frac{1}{n}\right)^{(1/11)\epsilon^2 k'}$.

We now bound the probability that $\min(G_2\lceil_\rho)$ is not k'-coverable. For all pairs (i, j), $i \in D_0$, $j \in D_1$, construct the formulas $\phi_{ij} = \bigvee\{\alpha : \alpha \text{ is a map in } G_2 \text{ containing } P_{ij}\}$.

Then construct the formulas $\phi'_{ij} = \bigvee\{\alpha' : (\alpha' \bigwedge P_{ij})$ is a map in $\phi_{ij}\}$, where ϕ'_{ij} is ϕ_{ij} with P_{ij} removed.

We now proceed in three phases: in Phase 0 we use the induction hypothesis on each ϕ'_{ij} to obtain sets C'_{ij} covering the formulas $\min(\phi'_{ij}\lceil_\rho)$; in Phase 1 we apply Lemma 5.2 to the systems $\{j \to C'_{ij}\}$ for each i to obtain sets C'_i; and in Phase 2 we apply Lemma 5.2 once more to the system $\{i \to C'_i\}$. The resulting set covers every $\min(\phi'_{ij}\lceil_\rho)$; by an argument using the fact that every map in G_2 has size at least 2, the set also covers every $\min(\phi_{ij}\lceil_\rho)$ and, therefore, covers $\min(G\lceil_\rho)$.

In Phases 0, 1, and 2 we apply successive restrictions $\rho_0 \in \Omega_{n_0,\epsilon^{2(t-1)}}$, $\rho_1 \in \Omega_{n_1,\epsilon}$, and $\rho_2 \in \Omega_{n_2,\epsilon}$ whose composition is the restriction ρ required for the lemma. Here we define $n_0 = n = |D_0|$, $n_1 = n^{\epsilon^{2t-2}}$, and $n_2 = n^{\epsilon^{2t-1}}$, corresponding to the domain size remaining before each of the three phases. The domains themselves we denote by $(D_0^0, D_1^0) = (D_0, D_1)$, (D_0^1, D_1^1), and (D_0^2, D_1^2), reserving (D_0^3, D_1^3) for the domain at the end of Phase 2. Among all the formulas ϕ_{ij}, the only ones that will ultimately be significant are those for which $i \in D_0^3$ and $j \in D_1^3$, since a cover of these formulas after applying ρ is sufficient to cover G after applying ρ.

Phase 0. Each of the $n_0{}^2 - n$ different formulas ϕ'_{ij} is a $t-1$ disjunction. Applying the induction hypothesis, we have that a random restriction ρ_0 from $\Omega_{n_0,\epsilon^{2(t-1)}}$ has probability at most $n_0{}^2\alpha_l^{n_0,t-1}$ of failing to make all the formulas l-coverable for appropriately small l. Let C'_{ij} be a set of size l covering $\min(\phi'_{ij}\lceil_{\rho_0})$. We can assume that $i, j \notin C'_{ij}$ because every map in ϕ_{ij} contains P_{ij}, and, therefore, no map in ϕ'_{ij} contains any variable incident on $\{i, j\}$.

Fixing any δ such that $0 < \delta < \frac{1}{4}$, and choosing $l = n_1^{(1-\delta)}$, the required induction condition $l \le n^{\epsilon^{2(t-1)}}$ follows trivially. Finally, we can observe that $C'_{ij} \subseteq D_0^1 \cup D_1^1$ since the only variables set to $*$ by ρ_0 are those in $\{P_{xy} : x \in D_0^1 \text{ and } y \in D_1^1\}$.

Phase 1. By the choice of l, Lemma 5.2 can be applied to the n_1 different systems $S_i = \{j \to C'_{ij} : j \in D_1^1\}$, where $i \in D_0^1$. The required condition $x \notin g(x)$ follows since $j \notin C'_{ij}$ as noted above. We choose a single random restriction ρ_1 from $\Omega_{n_1,\epsilon}$ and obtain that the covering sets described in Lemma 5.2 fail to exist with probability at most $n_1\beta_\lambda^{n_1}$ for appropriately small λ. Thus with high probability we obtain sets C'_i, for $i \in D_0^1$, such that $C'_i = \cup_{j\in D_1^2}(C'_{ij} \cap (D_0^2 \cup D_1^2))$, and $|C'_i| \le \lambda$.

For each $i \in D_0^1$, C'_i covers every variable that is both covered by some C'_{ij} $(j \in D_1^1)$ and set to $*$ by ρ_1. Since C'_{ij} covers $\min(\phi'_{ij}\lceil_{\rho_0})$, this implies that C'_i covers $\min(\phi'_{ij}\lceil_{\rho_0})\lceil_{\rho_1}$ for $j \in D_1^1$. Hence C'_i covers $\min(\phi'_{ij}\lceil_{\rho_0\rho_1})$ for $i \in D_0^2$, $j \in D_1^2$. Choose $\lambda = n_2^{(1-\delta)}$.

Phase 2. By the choice of λ, Lemma 5.2 can be applied to the system $S = \{i \to C'_i : i \in D_0^2\}$. After choosing $\rho_2 \in \Omega_{n_2,\epsilon}$, the covering set described by Lemma 5.2 fails to exist with probability at most $\beta_{k'}^{n_2}$. Thus we probably obtain a set C' such that $C' = \cup_{i\in D_0^3}(C'_i \cap (D_0^3 \cup D_1^3))$ and $|C'| \le k'$.

The set C' covers every variable that is both covered by some C'_i $(i \in D_0^2)$ and set to $*$ by ρ_2. Since C'_i covers $\min(\phi'_{ij}\lceil_{\rho_0\rho_1})$, this implies that C' covers $\min(\phi'_{ij}\lceil_{\rho_0\rho_1})\lceil_{\rho_2}$. Hence C' covers $\min(\phi'_{ij}\lceil_{\rho_0\rho_1\rho_2})$ for $i \in D_0^3$, $j \in D_0^3$.

It remains to argue that C' covers $\min(\phi_{ij}\lceil_\rho)$ for $i \in D_0^3$, $j \in D_1^3$ (recall $\rho = \rho_0\rho_1\rho_2$). Observe that every $\bigwedge$-clause of $\min(\phi_{ij}\lceil_\rho)$ is a clause of $\min(\phi'_{ij}\lceil_\rho)$, possibly with P_{ij} added.

Proof. Consider cases in which $\rho_0(P_{ij}) = 0$, 1, or $*$. If $\rho(P_{ij}) = *$, then P_{ij} appears in every clause of $\phi_{ij}\lceil_\rho$; therefore, clauses eliminated from $\phi'_{ij}\lceil_\rho$ by the min operator will also be eliminated from $\phi_{ij}\lceil_\rho$. Therefore, we already have the fact that for all $i \in D_0^3$, $j \in D_1^3$, the set C' covers all the variables in $\phi_{ij}\lceil_\rho$ except possibly P_{ij}.

Since maps in ϕ_{ij} have size at least 2, there is a variable P_{kl} in ϕ_{ij} with $k \neq i$ and $l \neq j$. By definition, P_{ij} is a variable in ϕ_{kl} and, therefore, is included in the fact regarding ϕ_{kl}.

Analysis for bounding G_2. The total probability that we fail to obtain the covering set is at most the sum of the probabilities in the three phases. This amount is $n_0{}^2\alpha_l^{n,t-1} + n_1\beta_\lambda^{n_1} + \beta_{k'}^{n_2}$. Using the constraint on $k(=2k')$ from the statement of the lemma, it can be seen that $\beta_{k'}^{n_2}$ is the dominant term in this sum. The amount is

$$n_0{}^2\left(\frac{1}{n_0}\right)^{\left(\frac{1}{11}\epsilon^{2t-2}n_1^{(1-\delta)}\right)} + n_1\left(\frac{1}{n_1}\right)^{\left(\frac{1}{5}\epsilon n_2^{(1-\delta)}\right)} + \left(\frac{1}{n_2}\right)^{\left(\frac{1}{5}\epsilon k'\right)}$$
$$= \left(\frac{1}{n}\right)^{\left(\frac{1}{11}\epsilon^{2t-2}n^{(1-\delta)\epsilon^{2t-2}}-2\right)} + \left(\frac{1}{n}\right)^{\left(\frac{1}{5}\epsilon^{2t-1}n^{(1-\delta)\epsilon^{2t-1}}-\epsilon^{2t-2}\right)} + \left(\frac{1}{n}\right)^{\left(\frac{1}{5}\epsilon^{2t}k'\right)}$$
$$\leq 3\left(\frac{1}{n}\right)^{\left(\frac{1}{5}\epsilon^{2t}k'\right)}.$$

The last inequalities are obtained as follows. The condition $t \in o(\log\log n)$ implies that $n^{\epsilon^{2t}}$ is increasing (i.e., $\omega(1)$); therefore, the conditions $k' \leq n^{\epsilon^{2t}}$ and $(1-\delta)/\epsilon > 1$ imply that

$$k' \leq \frac{1}{\epsilon}\left(n^{\epsilon^{2t}}\right)^{(1-\delta)/\epsilon} - \frac{1}{\epsilon}$$

and

$$k' \leq \frac{5}{11\epsilon^2}\left(n^{\epsilon^{2t}}\right)^{(1-\delta)/(\epsilon^2)} - \frac{2}{\epsilon^{2t}}.$$

These two inequalities imply that the third exponent above, $(\frac{1}{5}\epsilon^{2t}k')$, is smaller than the other two.

Combining the bounds for G_1 and G_2. The overall probability is at most the sum of the probabilities for G_1 and G_2:

$$\left(\frac{1}{n}\right)^{\frac{1}{11}\epsilon^2 k'} + 3\left(\frac{1}{n}\right)^{\left(\frac{1}{5}\epsilon^{2t}k'\right)}$$
$$\leq \left(\frac{1}{n}\right)^{\frac{1}{22}\epsilon^2 k} + 3\left(\frac{1}{n}\right)^{\left(\frac{1}{10}\epsilon^{2t}k\right)}$$
$$\leq \left(\frac{1}{n}\right)^{\left(\frac{1}{11}\epsilon^{2t}k\right)}$$
$$\leq \alpha_k^{n,t}.$$

The third line follows from the second line because for $\epsilon < \frac{1}{16}$ and $t \geq 2$, the exponent in the second term $(\frac{1}{10}\epsilon^{2t}k)$ is smaller than the exponent in the first term $(\frac{1}{22}\epsilon^2 k)$; hence the second term is larger than the first term. (And twice the second term is easily dominated by the term in the third line.) □

The following lemma states that if g is a function taking $x \in D_0 \cup D_1$ to a small subset of $D_0 \cup D_1$ not containing x, then g, when restricted to a random subset of size n^ϵ, will have a small sized range. Recall that $n = |D_1| = |D_0| - 1$.

LEMMA 5.2. *Let $0 < \delta < 1/3$, $0 \leq \epsilon \leq \delta^2/4$, and let g be a function defined on $D_0 \cup D_1$, such that $g(x) \subseteq D_0 \cup D_1$, $|g(x)| \leq n^{1-\delta}$, and $x \notin g(x)$ for all $x \in D_0 \cup D_1$. Then for all $t > 4/\sqrt{\epsilon}$ and for all sufficiently large n we have*

$$Pr^{n,\epsilon}_{\langle r,s\rangle}[\left|\bigcup_{x\in s} g(x) \cap s\right| > t] \leq \beta^n_t,$$

where

$$\beta^n_t = n^{-t(\frac{\epsilon}{5})}.$$

We will need the following lemmas, based on Ajtai's (*) and (**), to prove Lemma 5.2.

LEMMA 5.3. *Let A be a set of subsets of $D_0 \cup D_1$ such that $|A| \leq n^c$, and $|X| < n^{1-\delta}$, for all $X \in A$. Then for all $t > 0$ and for all sufficiently large n,*

$$Pr^{n,\epsilon}_{\rho=<r,s>}[\exists X \in A,\ |X \cap s| \geq t] \leq 2n^{-\frac{t}{2}(\delta-\epsilon)+c}.$$

Proof. For $t \geq n^\epsilon$ the above probability is zero, so we can assume that $t \leq n^\epsilon$. If there exists an $X \in A$ such that the event $|X \cap s| \geq t$ holds, then either (1) $|X \cap s_0| \geq t/2$ or (2) $|X \cap s_1| \geq t/2$. To bound (2), consider a fixed subset X of A. The probability that a random set s_1 contains at least $t/2$ elements of X is no greater than $[\binom{|X|}{t/2}\binom{n-t/2}{|s_1|-t/2}]/[\binom{n}{|s_1|}]$. Because there are at most n^c subsets $X \in A$, the probability that a random s_1 contains at least $t/2$ elements of some X is at most

$$\begin{aligned}
&n^c\binom{n^{1-\delta}}{t/2}\binom{n-t/2}{n^\epsilon - t/2}\Big/\binom{n}{n^\epsilon}\\
&= \frac{n^c(n^{1-\delta})!(n-t/2)!(n-n^\epsilon)!(n^\epsilon)!}{(n^{1-\delta}-t/2)!t/2!(n-n^\epsilon)!(n^\epsilon-t/2)!n!}\\
&\leq \frac{n^c \cdot n^{(1-\delta)t/2} \cdot n^{(\epsilon)t/2}}{n \cdot 2(n-1)\cdot 3(n-2)\cdot \ldots \cdot (t/2)(n-t/2+1)}\\
&\leq \frac{n^c \cdot n^{(1+\epsilon-\delta)t/2}}{n^{t/2}} \quad \text{because } k(n-k+1) \geq n \quad \text{for all } k \leq n\\
&= n^{-\frac{t}{2}(\delta-\epsilon)+c}.
\end{aligned}$$

It is not too hard to show that the above inequalities also hold for s_0. Therefore, $\Pr^{n,\epsilon}_{\langle r,s\rangle}[\exists X \in A,\ |X \cap s| > t] \leq 2n^{-\frac{t}{2}(\delta-\epsilon)+c}$. □

LEMMA 5.4. *Let g' be a function defined on $D_0 \cup D_1$ such that $g'(x) \subseteq D$, $|g'(x)| \leq t$, and $x \notin g'(x)$ for every $x \in D_0 \cup D_1$. Then for all $t > 0$ and for all sufficiently large n,*

$$Pr^{n,\epsilon}_{\langle r,s\rangle}[|\{y : \exists x \in s,\ y \in g'(x) \cap s\}| > t] \leq 2n^{-\frac{t}{4}(1-4\epsilon)}.$$

Proof. Let elements of s be obtained by choosing the elements of s_0 without replacement from D_0, and then choosing the elements of s_1 without replacement from

D_1. Letting $q = 2n^\epsilon + 1 = |s|$, label the chosen elements of s by $\{1, \cdots, q\}$ in the order in which they were chosen.

Intuitively, we visualize the function $g'(x)$ on the set s by a directed graph with q nodes, ordered in a line and labelled by elements of s; the edge (i, j) is present if and only if $j \in g'(i)$. If the condition of the lemma holds, then either $t/2$ edges point to the left or $t/2$ edges point to the right. More formally, if $|\{y : \exists x \in s,\ y \in g'(x) \cap s\}| > t$, then either (1) $|\{y : \exists x \in s,\ y \in g'(x) \cap s \text{ and } x < y\}| > t/2$ or (2) $|\{y : \exists x \in s,\ y \in g'(x) \cap s \ \&\ x > y\}| > t/2$.

We analyze case (1); case (2) is similar. Suppose H is a set of size $t/2$ that satisfies (1). By definition, (3) $\forall y \in H,\ y \in \bigcup_{1 \le x < y} g'(x)$. We first upper-bound the probability of (3) for a particular subset H, and then sum the probability over all possibilities for H.

When we pick the yth element of s, the set $\bigcup_{1 \le x < y} g'(x)$ has at most ty elements, and there are at least $n - y + 1$ elements of D_0 (or D_1) to choose from. Therefore, the probability that y is in the set H is at most $(\frac{ty}{n-y+1})$, which is at most $(\frac{2qt}{n})$, since $y \le q$ and $n/2 \ge q - 1$ for sufficiently large n.

Since the number of possible subsets H is $\binom{q}{t/2}$, the probability of (3) is at most $(\frac{2qt}{n})^{t/2}\binom{q}{t/2}$. Defining τ by $t = n^\tau$ and estimating $2q \le n^{2\epsilon}$, we get

$$\left(\frac{2qt}{n}\right)^{\frac{t}{2}} \binom{q}{t/2} \le n^{\frac{t}{2}(\tau+2\epsilon-1)} n^{\frac{t}{2}(2\epsilon)}/(t/2)! = n^{\frac{t}{2}(\tau-(1-4\epsilon))}/(t/2)!.$$

Now there are two cases. If $\tau < (1-4\epsilon)/2$, then the probability is smaller than $n^{-(t/4)(1-4\epsilon)}$. On the other hand, if $\tau \ge (1 - 4\epsilon)/2$, then t is a small polynomial in n; using the Stirling approximation,

$$\left(\frac{t}{2}\right)! \ge \left(\frac{t}{2}\right)^{\frac{t}{2}(1-\frac{\log e}{\log(t/2)})} \ge \left(n^{\tau-(2/\log n)}\right)^{\frac{t}{2}(1-\frac{\log e}{\log(t/2)})} \ge n^{\frac{t}{2}(\tau-(1-4\epsilon)/2)}.$$

In this case, the probability is also smaller than $n^{-\frac{t}{4}(1-4\epsilon)}$. Combining this estimate for cases (1) and (2), the overall probability is no greater than $2n^{-\frac{t}{4}(1-4\epsilon)}$. □

Proof of Lemma 5.2. To prove Lemma 5.2, we apply two successive restrictions, $\rho^1 = \langle r^1, s^1\rangle \in \Omega^{n,\sqrt{\epsilon}}$ and $\rho^2 = \langle r^2, s^2\rangle \in \Omega^{n^{\sqrt{\epsilon}},\sqrt{\epsilon}}$. First applying Lemma 5.3 for a random $\rho^1 \in \Omega^{n,\sqrt{\epsilon}}$, with $A = \{g(x)|x \in D\}$ and $c = 1$ we obtain

$$\Pr^{n,\sqrt{\epsilon}}_{\langle r^1,s^1\rangle}[\exists x \in s^1\ |g(x) \cap s^1|\ > t] \le 2n^{-\frac{t}{2}(\delta-\sqrt{\epsilon})+1} \le 2n^{-\frac{t\sqrt{\epsilon}}{2}+1} \le 2n^{-\frac{t\sqrt{\epsilon}}{4}}.$$

The second inequality holds because $\delta \ge 2\sqrt{\epsilon}$; the last inequality holds because for $t > \frac{4}{\sqrt{\epsilon}}$, $\frac{t\sqrt{\epsilon}}{2} > 2$.

Assuming that the first restriction is successful (i.e., $\forall x \in D$, $|g(x) \cap s^1| < t$), we can define $g'(x) = g(x) \cap s^1$ and apply Lemma 5.4 on the domain $D_0 \cap s^1$, $D_1 \cap s^1$, drawing a random $\rho^2 = \langle r^2, s^2\rangle \in \Omega^{n^{\sqrt{\epsilon}},\sqrt{\epsilon}}$, to get

$$\Pr^{n^{\sqrt{\epsilon}},\sqrt{\epsilon}}_{\langle r^2,s^2\rangle}[|\{y : \exists x \in s^2 y \in g'(x) \cap s^2\}| > t] \le 2\left(n^{\sqrt{\epsilon}}\right)^{-\frac{t}{4}(1-4\sqrt{\epsilon})} \le 2n^{-t\epsilon/4}.$$

The last inequality holds because for $\epsilon \le \frac{1}{25}$, $\sqrt{\epsilon} - 4\epsilon \ge \epsilon$.

Now, a randomly chosen $\rho \in \Omega^{n,\epsilon}$ can be viewed as the composition of the two random choices for ρ^1 and ρ^2. Thus the probability that a random ρ satisfies the condition of Lemma 5.2 is at most $2n^{-\frac{t\sqrt{\epsilon}}{4}} + 2n^{-\frac{t\epsilon}{4}} \le 4n^{-\frac{t\epsilon}{4}} \le n^{-\frac{t\epsilon}{5}}$ for n sufficiently large. □

6. Limitations of the covering lemma. The covering lemma states that with high probability any given t-disjunction will be covered by a set of size k, after $O(t)$ restrictions. Because of the large number of restrictions that must be applied for every application of the covering lemma, the map size, t, cannot be too large (otherwise, we quickly end up with an assignment to all of the variables). Therefore, one way of improving the bound would be to prove the covering lemma for a *single* restriction. This stronger form of the covering lemma could be stated as: For any t, $\epsilon < 1$, and for any t-disjunction, G, $\Pr_\rho^{n,\epsilon}[\text{Cover}(\min(G\lceil_\rho)) > k] \leq \alpha^k$, for some $\alpha < 1$, where α depends possibly on n and t. Setting $t = k$ approximately equal to $n^{1/d}$, this strengthened covering lemma would yield an exponential lower bound for PHP_n.

Unfortunately, this strengthened version is false for $k > \log n$. This situation is similar to the impossibility of obtaining an exponential lower bound for bounded depth circuits computing the parity function by simply improving the combinatorial lemma in [FSS]. Here we briefly describe a function, due to Russell Impagliazzo, which contradicts this strengthened covering statement for $t = \log n + 1$.

The multiplexor function is a function on $n + \log n$ bits, $\{x_k\}$, where the first $\log n$ bits are used to index the remaining n bits. The function is "1" if and only if the value indexed by the first $\log n$ bits is "1." This function can be written as the OR of n minterms, each of size $\log n + 1$.

The counterexample to the strengthened covering lemma is a t-disjunction, which encodes the multiplexor function on $\{x_k\}$ using the pigeonhole variables P_{ij}. Because the new function has to be monotone, we will encode negation by using the range elements, D_1. The "pigeonhole" multiplexor function is a function on variables $\{P_{ij} \mid i \in D_0,\ j \in D_1\}$, where $|D_0| = \log n + n + 1$, and $|D_1| = \log n + n$. Let T be a fixed subset of D_1 of size $|D_1|/2$. An assignment ρ for $\{P_{ab}\}$ which is one-one on D_1 induces an assignment to the $n + \log n$ variables $\{x_i\}$ by $x_i = 1$ if and only if $\exists j \in T$ such that $\rho(P_{ij}) = 1$. The value of the pigeonhole multiplexor function is the value of the multiplexor function on these induced values. Note that the modified function can be written as a t-disjunction for $t = \log n + 1$.

Let $\rho =< r, s >$ be a random restriction from $\Omega^{n,\epsilon}$. Intuitively, if all of the $\log n$ index variables in D_0 are included in s, then the restricted function will have large covering sets. The probability of this happening is approximately $(\frac{n^\epsilon}{n})^{\log n}$, which is larger than α^k for $\alpha < 1$ and $k = n^{1/2}$.

This counterexample shows that an exponential lower bound for PHP_n cannot be obtained by simply improving the covering lemma. However, we feel that the covering lemma can be improved to yield a lower bound of $n^{\log^c n}$, for a small constant c, independent of the depth.

7. Conclusions and open problems. In this paper we have given a proof-theoretic superpolynomial lower bound for constant depth Frege proofs of the pigeonhole principle. Our approach introduces the notion of using approximations for a sequence of formulas, and shows how to use a proof theoretic approach to eliminate the nonstandard model theory that was used by Ajtai. We also improve the lower bound of Ajtai. In addition, this proof more directly explains why bounded depth Frege proofs are weak for proving the pigeonhole principle.

We avoid the nonconstructivity of the compactness theorem; in fact, it appears likely [P] that our proof can be made feasibly constructive as defined in [CU]. Informally, a feasibly constructive lower bound proof is one which involves only polynomial-time concepts. In contrast, it was shown in [CU] that a superpolynomial lower bound for *extended* Frege systems cannot have a feasibly constructive proof. A formalization of our result

as a feasibly constructive proof requires describing exactly how to choose the restrictions, using Spencer's "probabilistic method" for transforming probabilistic algorithms into deterministic ones [Sp, p. 31].

An outstanding open question is to prove a truly exponential lower bound for bounded depth Frege proofs. Such a bound would imply that S_2 (a subsystem of Peano arithmetic containing $I\Delta_0$) augmented by a function symbol f cannot prove the sentence asserting PHP for f. (See [PWW], [Bu] for the connection between subsystems of bounded arithmetic and bounded depth Frege proofs.) As it is, current results simply imply that $I\Delta_0(f)$ cannot prove PHP(f). It is known [PWW] that $I\Delta_0(f)$, together with the existence of the function $n^{\log n}$, can prove the weak pigeonhole principle for f, i.e., the principle that f is not a bijection between $[2n]$ and $[n]$.

Acknowledgments. We wish to thank Stephen Cook, Russell Impagliazzo, and Alan Woods for many valuable conversations that led to this proof. We thank Paul Beame for his help in correcting the covering lemma. We thank Alexander Razborov and Judy Goldsmith for their comments on earlier drafts of the paper.

REFERENCES

[Ajt] M. AJTAI, *The complexity of the pigeonhole principle*, Proceedings of the 29th Annual IEEE Symposium on the Foundations of Computer Science, White Plains, NY, 1988, pp. 346–355; preliminary version.

[Ajt2] ———, *Σ_1^1-formulae on finite structures*, Ann. Pure Appl. Logic, 24 (1983), pp. 1–48.

[Ajt3] ———, *First order definability on finite structures*, Ann. Pure Appl. Logic, 45 (1989), pp. 211–225.

[Bu] S. BUSS, *Polynomial size proofs of the propositional pigeonhole principle*, J. Symbolic Logic, 52 (1987), pp. 916–927.

[CR] S. A. COOK AND R. RECKHOW, *The relative efficiency of propositional proof systems*, J. Symbolic Logic, 44 (1979), pp. 36–50.

[CU] S. A. COOK AND A. URQUHART, *Functional interpretations of feasibly constructive arithmetic*, Tech. Report 210/88, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1988.

[FSS] M. FURST, J. SAXE, AND M. SIPSER, *Parity, circuits and the polynomial time hierarchy*, Math. Systems Theory, 17 (1984), pp. 13–27.

[Ha] A. HAKEN, *The intractability of resolution*, Theoret. Comput. Sci., 39 (1985), pp. 297–308.

[H] J. HÅSTAD, *Computational Limitations of Small-Depth Circuits*, MIT Press, Cambridge, MA, 1987.

[P] T. PITASSI, *A feasibly constructive lower bound for parity*, manuscript, 1990.

[PW] J. PARIS AND A. WILKIE, *Counting problems in bounded arithmetic*, in Methods in Mathematical Logic, *Proceedings of the Sixth Latin American Symposium on Mathematical Logic*, Caracas, 1983; Lecture Notes in Math., 1130, Springer-Verlag, Berlin, 1985, pp. 317–340.

[PWW] J. PARIS, A. WILKIE, AND A. WOODS, *Provability of the pigeonhole principle and the existence of infinitely many primes*, J. Symbolic Logic, 53 (1988), pp. 1235–1244.

[Raz] A. A. RAZBOROV, *Lower bounds on the monotone complexity of some Boolean functions*, Soviet Math. Dokl., 31 (1985), pp. 354–357.

[Sh] J. SHOENFIELD, *Mathematical Logic*, Addison-Wesley, Reading, MA, 1967.

[Si] M. SIPSER, *Borel sets and circuit complexity*, Proceedings of the 15th ACM Symposium on Theory of Computing, Boston, MA, 1983, pp. 330–335.

[Sp] J. SPENCER, *Ten Lectures on the Probabilistic Method*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987.

[Urq] A. URQUHART, *Hard examples for resolution*, J. Assoc. Comput. Mach., 34 (1987), pp. 209–219.

[Y] A. YAO, *Separating the polynomial-time hierarchy by oracles*, Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science, Portland, OR, 1985, pp. 1–10.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1180–1183, December 1992
© 1992 Society for Industrial and Applied Mathematics
011

EXTERNAL INTERNAL NODES IN DIGITAL SEARCH TREES VIA MELLIN TRANSFORMS*

HELMUT PRODINGER†

Abstract. Knuth posed it as an open problem to compute the average number of external internal nodes in a random digital search tree. This was settled by Flajolet and Sedgewick using Rice's method. This note offers an alternative approach using Mellin transforms.

Key words. digital search trees, Mellin transform

AMS(MOS) subject classifications. 68Q25, 68P05, 68R10

Knuth [4, Ex. 6.3.29] posed it as an open problem (ranked $M46$) to compute (asymptotically) l_N, the average number of external internal nodes in a random *digital search tree* built by N data.

This problem was settled by Flajolet and Sedgewick [2] by a very convenient technique called *Rice's method*. This method has some advantages over the *Mellin transform technique* that Knuth used heavily in his famous book. (In passing it will be mentioned that the variance was treated by Kirschenhofer and the author [3].)

In this note we show that *Knuth could have given a solution himself* because in the present approach we follow his ideas that he used in a similar (although easier) problem (cf. [4, p. 497]).

For a complete description of the problem and digital search trees we must refer to [4] and [2] in order to keep this note short. For the use of the Mellin transform technique we cite [1].

The problem is to compute

$$l_N = N - \sum_{k=2}^{N} \binom{N}{k} (-1)^k R_{k-2}$$

with

$$R_N = Q_N \sum_{k=0}^{N} \frac{1}{Q_k}$$

and

$$Q_k = \left(1 - \frac{1}{2}\right)\left(1 - \frac{1}{4}\right)\dots\left(1 - \frac{1}{2^k}\right) \quad \text{for } k \geq 1 \quad \text{and} \quad Q_0 = 1.$$

It follows from [2] and [3] that

$$R_N = N + 1 - \alpha + R^*(N)$$

with

$$\alpha = \sum_{k \geq 1} \frac{1}{2^k - 1} = 1.606695\dots$$

*Received by the editors August 21, 1991; accepted for publication (in revised form) January 13, 1992.
†Department of Algebra and Discrete Mathematics, Technical University of Vienna, Vienna, Austria.

and

$$R^*(z) = \frac{1}{Q_\infty} \sum_{j \geq 1} a_j \frac{z+j}{2^{z+j}-1}.$$

The quantities in the last equation are

$$Q_\infty = \lim_{k \to \infty} Q_k = 2.88788\ldots$$

and

$$a_j = (-1)^{j-1} 2^{-\binom{j}{2}} / Q_{j-1}.$$

We mention for later use that

$$R^*(-1) = \frac{1}{Q_\infty} \left[\frac{1}{\log 2} + \sum_{j \geq 2} a_j \frac{j-1}{2^{j-1}-1} \right] = 2.2346464\ldots$$

(a very fast converging series!).

Therefore,

$$l_N \sim N(\alpha + 1) - \Sigma$$

with

$$\Sigma = \frac{1}{Q_\infty} \sum_{j \geq 1} a_j \sum_{k=2}^{N} \binom{N}{k} (-1)^k \frac{k-2+j}{2^{k-2+j}-1},$$

and this sum Σ will be attacked in the style of Knuth. Expanding the geometric series, we get

$$\begin{aligned}
\Sigma &= \frac{1}{Q_\infty} \sum_{j \geq 1} a_j \sum_{m \geq 1} \sum_{k=2}^{N} \binom{N}{k} (-1)^k (k-2+j) 2^{-m(k-2+j)} \\
&= \frac{1}{Q_\infty} \sum_{j \geq 1} a_j \sum_{m \geq 1} \Bigg[-N 2^{-m(j-1)} \left[(1-2^{-m})^{N-1} - 1 \right] \\
&\qquad + (j-2) 2^{-m(j-2)} \left[(1-2^{-m})^N - 1 + \frac{N}{2^m} \right] \Bigg].
\end{aligned}$$

The last step was by the binomial theorem. Now let

$$\psi_1(x) = e^{-x} - 1 \quad \text{and} \quad \psi_2(x) = e^{-x} - 1 + x.$$

Then the sum Σ may be approximated by

$$\begin{aligned}
\Sigma &\sim \frac{1}{Q_\infty} \sum_{j \geq 1} a_j \sum_{m \geq 1} \left[-\frac{N}{2^{m(j-1)}} \psi_1\left(\frac{N}{2^m}\right) + \frac{j-2}{2^{m(j-2)}} \psi_2\left(\frac{N}{2^m}\right) \right] \\
&=: \Sigma_1 + \Sigma_2.
\end{aligned}$$

This is achieved by the *exponential approximation* $(1-a)^N \approx e^{-aN}$; see, e.g., [4; p. 131].

Let us start with Σ_1:

$$\Sigma_1 = -\frac{N}{Q_\infty} \cdot f_1(N)$$

with

$$f_1(x) = \sum_{j\geq 1} a_j \sum_{m\geq 1} 2^{-m(j-1)} \psi_1\left(\frac{x}{2^m}\right).$$

Its Mellin transform is for $-1 < \Re s < 0$,

$$\begin{aligned} f_1^*(s) &= \sum_{j\geq 1} a_j \sum_{m\geq 1} 2^{-m(j-1)} 2^{ms} \Gamma(s) \\ &= \Gamma(s) \cdot \sum_{j\geq 1} a_j \frac{1}{2^{j-1-s}-1}. \end{aligned}$$

The Mellin inversion formula

$$f(x) = \frac{1}{2\pi i} \int f^*(s) x^{-s} ds$$

(where the integration is along a vertical line inside the so-called *fundamental strip*) allows us to go back to the original function. To evaluate it asymptotically, we have to consider the negative residues of the integrand right of the fundamental strip (i.e., here at $s = 0$). We don't discuss the other residues which lead to smaller order terms, respectively, periodic fluctuations of small amplitude, but this can be done as easily as the main term.

The term $j = 1$ must be treated separately, as it contains a *second-order pole*. The residue computation is standard and gives a contribution

$$-\frac{N}{Q_\infty}\left(-\log_2 N + \frac{1}{2} - \frac{\gamma}{\log 2}\right)$$

to Σ_1. For $j \geq 2$, there is only a *simple pole*, yielding a contribution

$$-\frac{N}{Q_\infty}\left[-\sum_{j\geq 2} a_j \frac{1}{2^{j-1}-1}\right].$$

Now we turn to Σ_2:

$$\Sigma_2 = \frac{1}{Q_\infty} f_2(N)$$

with

$$f_2(x) = \sum_{j\geq 1} a_j (j-2) \sum_{m\geq 1} 2^{-m(j-2)} \psi_2\left(\frac{x}{2^m}\right).$$

We transform $f_2(x)$: (for $-2 < \Re s < -1$)

$$f_2^*(s) = \Gamma(s) \cdot \sum_{j\geq 1} a_j (j-2) \frac{1}{2^{j-2-s}-1}.$$

This time the pole of interest is $s = -1$.

Again, $j = 1$ must be treated separately, yielding a contribution

$$\frac{N}{Q_\infty}\left(-\log_2 N + \frac{1}{2} + \frac{1}{\log 2} - \frac{\gamma}{\log 2}\right)$$

to Σ_2. For $j \geq 2$, the contribution to Σ_2 is

$$\frac{N}{Q_\infty}\sum_{j\geq 2} a_j(j-2)\frac{1}{2^{j-1}-1}.$$

Hence we have

$$\begin{aligned}\Sigma_1 + \Sigma_2 &\sim \frac{N}{Q_\infty}\left(\frac{1}{\log 2} + \sum_{j\geq 2} a_j(j-1)\frac{1}{2^{j-1}-1}\right)\\ &= NR^*(-1).\end{aligned}$$

Therefore, we obtain the final result:

$$l_N \sim N\big(\alpha + 1 - R^*(-1)\big) = 0.37204\ldots \cdot N,$$

where, again, the *ubiquitous* fluctuating term of order N was not mentioned for brevity.

REFERENCES

[1] P. Flajolet, M. Régnier, and R. Sedgewick, *Some Uses of the Mellin Integral Transform in the Analysis of Algorithms*, in Combinatorics on Words, Springer NATO ASI Series F, Vol. 12, Berlin, 1985.

[2] P. Flajolet and R. Sedgewick, *Digital search trees revisited*, SIAM J. Comput., 15 (1986), pp. 748–767.

[3] P. Kirschenhofer and H. Prodinger, *Einige Anwendungen der Modulfunktionen in der Informatik*, Sitzungsber. Österreich. Akad. Wiss., 197 (1988), pp. 339–366.

[4] D. E. Knuth, *The Art of Computer Programming*, *Vol.* 3, Addison Wesley, Reading, MA, 1973.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1184–1192, December 1992
© 1992 Society for Industrial and Applied Mathematics
012

VERIFICATION AND SENSITIVITY ANALYSIS OF MINIMUM SPANNING TREES IN LINEAR TIME*

BRANDON DIXON[†‡], MONIKA RAUCH[†§], AND ROBERT E. TARJAN[†¶]

Abstract. Komlós has devised a way to use a linear number of binary comparisons to test whether a given spanning tree of a graph with edge costs is a minimum spanning tree. The total computational work required by his method is much larger than linear, however. This paper describes a linear-time algorithm for verifying a minimum spanning tree. This algorithm combines the result of Komlós with a preprocessing and table look-up method for small subproblems and with a previously known almost-linear-time algorithm. Additionally, an optimal deterministic algorithm and a linear-time randomized algorithm for sensitivity analysis of minimum spanning trees are presented.

Key words. network optimization, minimum spanning tree, program checking, sensitivity analysis, divide and conquer

AMS(MOS) subject classifications. 05, 68

1. Introduction. Suppose we wish to solve some problem for which we know in advance the size of the input data, using an algorithm from some well-defined class of algorithms. For example, consider sorting n numbers, when n is fixed in advance, using a binary comparison tree. Given a sufficient amount of preprocessing time and storage space, we can in a preprocessing step compute a minimum-depth comparison tree, store it explicitly, and then solve any instance of the sorting problem by using the precomputed comparison tree.

This technique is of course generally useless because it is prohibitively expensive in preprocessing time and storage space, both being at least exponential in n. There are situations in which this idea can be used to advantage, however. This is the case in problems susceptible to very efficient divide-and-conquer. The idea is to split the problem to be solved into subproblems, which are categorized into classes. If the subproblems are small enough, they can be solved efficiently as follows: An optimal algorithm for each class is precomputed and stored in a look-up table, and each instance of a subproblem is solved by looking up and running the algorithm for its class. For this technique to pay off, solving all the subproblems must reduce the original problem sufficiently so that it can be solved quickly with respect to the size of the original problem by using a nonoptimal algorithm.

This paper presents an application of this general technique to two problems concerning minimum spanning trees. This approach was first proposed explicitly by Larmore [15], who used it to solve a convex matrix searching problem. Related techniques were used by Gabow and Tarjan [8] to solve a disjoint set union problem, by Harel and Tarjan [12] to find nearest common ancestors in a tree, and by Fredman [5] to solve the all pairs shortest path problem.

---

*Received by the editors October 22, 1990; accepted for publication (in revised form) August 13, 1991.

†Department of Computer Science, Princeton University, Princeton, New Jersey 08544.

‡The research of the first author was partially supported by a National Science Foundation graduate fellowship.

§The research of the second author was supported by the German Fellowship Foundation, Studienstiftung des deutschen Volkes.

¶The research of the third author was conducted at NEC Research Institute, Princeton, New Jersey 08540. This research was partially supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center grant NSF-STC88-09648, and the Office of Naval Research contract N00014-87-K-0467.

We present an algorithm that verifies a minimum spanning tree in an n-vertex, m-edge graph in $O(m)$ time. We also give algorithms performing sensitivity analysis of minimum spanning trees in worst-case time minimum to within a constant factor and in linear expected time. Our model of computation allows edge costs to be compared, added, or subtracted at unit cost, and side computations to be performed on a unit-cost random-access machine with word size $\Omega(\log n)$ bits. The verification algorithm uses the comparison bound of Komlós [14] for the subproblems and Tarjan's $O(m\alpha(m,n))$ algorithm [17] for the reduced problem. For sensitivity analysis we solve the subproblems using a result of Goddard, King, and Schulman [9] in the randomized case and enumeration of all possible algorithms in the deterministic case. In both cases, Tarjan's $O(m\alpha(m,n))$-time sensitivity analysis algorithm [18] processes the reduced problem. We describe the algorithms in §§2 and 3. Section 4 contains concluding remarks.

It is important to note that our computational model allows only very simple operations on the edge costs, namely, binary comparison, addition, and subtraction. In a more powerful model in which bit manipulation of the edge costs is possible, better results can be obtained using table look-up and other ideas. In particular, Harel [11] showed that if the edge costs are integers polynomially bounded in n, then on a unit-cost random-access machine the minimum spanning tree verification and sensitivity analysis problems can be solved deterministically in linear time. More recently, Fredman and Willard [6] showed that if the edge costs are single-precision integers, then a minimum spanning tree can be computed deterministically in linear time on a unit-cost random access machine. These results are incomparable with ours because of the stronger computation model used.

2. Verification of minimum spanning trees. Let $G = (V, E)$ be a connected, undirected graph with vertex set V of size n and edge set E of size m. Suppose every edge $\{v,w\} \in E$ has a real-valued *cost* $c(v,w)$. A *minimum spanning tree* of G is a spanning tree whose total edge cost is minimum. The *minimum spanning tree verification problem* is that of determining whether a specific spanning tree T is a minimum spanning tree. Since G is connected, $m \geq n-1$. To simplify time bounds, we assume that $m \geq n$; otherwise, G itself is a tree.

Several results concerning the minimum spanning tree verification problem are known. There are many efficient algorithms for *finding* a minimum spanning tree, given only the graph G and the edge costs; see the survey paper by Graham and Hell [10] or the monograph by Tarjan [19, Chap. 6]. The fastest known algorithm for finding a minimum spanning tree is that of Gabow et al. [7], which runs in $O(m \log \beta(m,n))$ time, where $\beta(m,n) = \min\{i \mid \log^{(i)} n \leq m/n\}$, and $\log^{(i)} n$ is defined recursively by $\log^{(0)} n = n$, $\log^{(i+1)} n = \log\log^{(i)} n$. The verification problem was considered by Tarjan [17] and subsequently by Komlós [14]. Tarjan proposed a verification algorithm running in $O(m\alpha(m,n))$ time, where α is a functional inverse of Ackermann's function. Komlós showed that a minimum spanning tree can be verified in $O(m)$ binary comparisons between edge costs. Unfortunately, his method requires nonlinear time to determine which comparisons to make. Here we describe an algorithm that verifies a minimum spanning tree in $O(m)$ time.

Let T be a spanning tree whose minimality we wish to test. For any pair of vertices v, w, we denote by $T(v,w)$ the path in T from v to w. T is minimum if and only if, for every nontree edge $\{v,w\}$, $c(v,w) \geq \max\{c(x,y) \mid \{x,y\} \in T(v,w)\}$. In order to efficiently verify this condition, we replace each nontree edge $\{v,w\}$ by a set of up to six replacement edges, each of cost $c(v,w)$. This replacement leaves invariant the

minimality of T. Edge replacement is a two-stage process. To begin the first stage, we choose an arbitrary vertex r and root T at r. We denote by $p(v)$ the parent of vertex v in the rooted version of T. For each nontree edge $\{v, w\}$, we compute the nearest common ancestor of v and w in T, say u. If v and w are unrelated in T (i.e., $u \notin \{v, w\}$), we replace $\{v, w\}$ by the pair of edges $\{u, v\}$, $\{u, w\}$, each with cost $c(v, w)$. Such replacement leaves invariant the minimality of T, at most doubles the number of nontree edges, and results in a graph such that every nontree edge joins two related vertices in T. The time to perform this replacement is $O(m)$ using either of the known linear-time algorithms for computing nearest common ancestors [12], [16].

We can now assume that each nontree edge $\{u, v\}$ is such that vertex u is an ancestor of vertex v. In the second stage, we replace each such edge by a set of up to three edges. In order to determine the edge replacements, we partition T into a collection of edge-disjoint subtrees. Let $g \geq 1$ be an integer parameter, whose value we shall specify later. The subtrees have two properties:

(i) There are at most $(n-1)/g + 1$ subtrees; and

(ii) Deletion from any subtree of its root and all edges incident to the root leaves a collection of smaller subtrees, called *microtrees*, each containing at most g vertices.

We compute the collection of subtrees in $O(n)$ time, as follows. We process all the vertices except r in postorder [19]. (This order guarantees that a parent is processed after all of its children.) When processing a vertex v, we compute an integer value $s(v)$ for it; and, in addition, we may mark it as a subtree root. The computed value of $s(v)$ is the number of descendants of v in T (including v itself) that are in the same microtree as v. Initially all vertices are unmarked. The vertex processing step is as follows:

$process(v)$: Compute $h = 1 + \sum\{s(w) \mid w \text{ is a child of } v\}$. If $h \leq g$, then let $s(v) = h$; otherwise, mark v as a subtree root, and let $s(v) = 1$.

Once the vertex processing is completed we mark r, the root of T, as a subtree root. Condition (ii) is immediate from the definition of the vertex processing. Condition (i) is also immediate: each subtree, except possibly the one rooted at r, contains more than g vertices and hence contains at least g edges, which means that there are at most $(n-1)/g + 1$ subtrees.

Let T' be the tree whose vertices are the marked vertices of T, with v the parent of w in T' if v is the deepest marked proper ancestor of w in T (i.e., the first marked vertex encountered on the path from w to r in T). We call T' the *macrotree*. By (i), T' has $O(n/g)$ vertices. Tree T' can be computed in $O(n)$ time by doing a depth-first traversal of T and maintaining the set of marked proper ancestors of the currently visited vertex on a stack; when the search visits a vertex v, the deepest marked proper ancestor of v, which we denote by $p'(v)$, is on top of the stack. (We adopt the convention that $p'(r)$ is undefined.)

We use the macrotree to define the replacement edges for each nontree edge. Let $\{u, v\}$ be such a nontree edge, with u an ancestor of v. Let $r_1 = p'(u)$ if u is unmarked or u if u is marked. Similarly, let $r_3 = p'(v)$ if v is unmarked or v if v is marked. If $r_1 = r_3$, we do not replace $\{u, v\}$. If $r_1 \neq r_3$, let r_2 be the child of r_1 in T' that is an ancestor of r_3; replace $\{u, v\}$ by $\{u, r_2\}$, $\{r_2, r_3\}$, $\{r_3, v\}$, deleting any of these edges that is a loop (an edge of the form $\{x, x\}$ for some x). Each new edge has a cost of $c(u, v)$. This replacement leaves invariant the minimality of T and at most triples the number of nontree edges.

We can compute the replacement edges for every nontree edge in a total of $O(m)$ time, as follows. A depth-first traversal of T as described above allows us to compute $p'(v)$ for each vertex $v \neq r$. This gives r_1 and r_3 in the edge replacement construction.

It remains to compute the r_2-vertices in the edge replacement construction. The computation of these vertices requires answering $O(m)$ queries of the following form on T': given a vertex z and another vertex y that is a proper ancestor of z, determine the child of y that is an ancestor of z. These queries can be answered in $O(m)$ time by performing a depth-first traversal of T', maintaining a stack of the ancestors of the currently visited vertex, and answering the query for a pair y, z when visiting z during the search, by reporting as the answer to the query the vertex just above y on the stack.

Having computed all the replacement edges, we must test, for each replacement edge $\{w, x\}$, whether $c(w, x) \geq \max\{c(y, z) \mid \{y, z\} \in T(w, x)\}$. In the rest of this section we describe how to perform this test for all replacement edges.

For each vertex $v \neq r$, we compute a value $high(v)$ equal to the maximum cost of an edge on the path $T(p'(v), v)$. These values can be computed for all vertices by doing a separate depth-first traversal of each of the subtrees of T that were determined by the partitioning process described previously. During the traversal of the subtree rooted at a vertex u, we maintain the path of edges from u to the currently visited vertex as a stack with heap order [19]; the values that are heap-ordered are the edge costs, and the *high*-values are computed using *find-max* operations. This data structure requires $O(1)$ amortized time per *push*, *pop*, or *find-max* operation [19]. Hence the total time to compute all *high*-values is $O(n)$. The *high*-values suffice to perform the required test for each of the $\{r_3, v\}$-replacement edges, in $O(1)$ time per edge: for such an edge, $high(v) = \max\{c(y, z) \mid \{y, z\} \in T(r_3, v)\}$.

We deal with the $\{r_2, r_3\}$-replacement edges by adding all of these edges to T' to form a graph G', giving each edge $\{p'(v), v\}$ in T' a cost $c(p'(v), v) = high(v)$, and verifying that T' is a minimum spanning tree in G'. To verify the minimality of T' we use the algorithm of Tarjan [17], which runs in $O(m\alpha(m, n'))$ time, where n' is the number of vertices in T'. If $g = \Omega(\log^{(i)} n)$ for any fixed positive integer i, then $n' = O(n/\log^{(i)} n)$, and $\alpha(m, n') = O(1)$ [19]. Thus verifying the minimality of T' takes $O(m)$ time.

The remaining edges that must be tested are the $\{u, r_2\}$-replacement edges. Each such edge has u and r_2 in the same microtree. Let $T_1, T_2, \ldots, T_k$ be the microtrees. For $1 \leq i \leq k$, we form a graph G_i by adding each $\{u, r_2\}$-replacement edge to the tree T_i such that u and r_2 are in T_i. Together the graphs $G_1, G_2, \ldots, G_k$ contain n vertices and $O(m)$ edges. By (ii), each G_i contains at most g vertices. We complete the task of verifying the minimality of T by verifying that T_i is a minimum spanning tree of G_i for each i in the range $1 \leq i \leq k$.

To verify the minimality of the microtrees, we use a preprocessing and table look-up technique. For each possible connected graph with no more than g vertices and specified spanning tree, we construct a short integer encoding by numbering the vertices consecutively from 1, encoding each edge by the pair of numbers of its end vertices, and concatenating the encodings of the edges, listing the spanning tree edges first. (It does not matter that this encoding is not unique.) The encoding for a graph-tree pair contains at most $\lceil \log g \rceil g^2/2$ bits, since there are fewer than $g^2/2$ edges. The total number of possible code strings (not all of which are legal encodings of graphs) is not more than $2^{\lceil \log g \rceil g^2/2}$. We will choose g such that each graph encoding fits into one computer word and such that there are at most $\sqrt{n}$ possible code strings. Choosing $g \leq c_2(\log n)^{1/3}$ for a suitably small value of c_2 more than suffices for this purpose.

Consider a connected graph with at most g vertices and $e < g^2/2$ edges and having a specified spanning tree T^*. The result of Komlós [14] implies that there is a decision tree D whose nodes represent binary comparisons of edge costs that will verify the minimality of T^* and has a depth of at most $c_1 e$, for some sufficiently large c_1. The number of nodes

in D is at most $2^{c_1 e+1}$. Furthermore, an inspection of the construction of Komlós shows that D can easily be constructed in $O(g^2)$ time per node, for a total of $O(g^2 2^{c_1 g^2/2})$ time.

Choosing $g \leq c_2(\log n)^{1/3}$ for a suitably small value of c_2 guarantees that the construction time for one decision tree is $O(\sqrt{n})$, and the total time required to construct decision trees for all possible graphs with at most g vertices is $O(n)$. Furthermore, the space needed to store all the decision trees is $O(n)$.

We construct one decision tree for each possible graph with at most g vertices and then build a table that maps code strings for graphs to the corresponding decision trees. Then we use the table to verify the minimality of the microtrees T_i in the respective graphs G_i, by computing a code string for each G_i, T_i pair, accessing the decision tree corresponding to the code string, and following the path through the decision tree determined by the edge costs of G_i. The total time to perform all the verifications is $O(m)$. This completes the verification of T.

The only constraints imposed on the choice of g in this construction are $g = \Omega(\log^{(i)} n)$ for some fixed positive integer i and $g \leq c_3(\log n)^{1/3}$ for $c_3 = \min\{c_0, c_2\}$. Thus it suffices to choose $g = c_3(\log n)^{1/3}$.

3. Sensitivity analysis of minimum spanning trees. An extension of the minimum spanning tree verification problem is the sensitivity analysis problem. Let G be an undirected graph with edge costs, and let T be a minimum spanning tree of G. The *sensitivity analysis problem* is to compute, for each edge $\{v, w\}$ of G, by how much $c(v, w)$ can change without affecting the minimality of G. Tarjan [18] has extended his verification algorithm to an algorithm that solves the sensitivity analysis problem in $O(m\alpha(m, n))$ time. For the special case of planar graphs, Booth and Westbrook [2] have given an algorithm running in $O(m)$ time. We shall describe a randomized $O(m)$-time algorithm and a deterministic algorithm that runs in time minimum to within a constant factor, although all that we can say for sure about the running time of the latter algorithm is that it is $O(m\alpha(m, n))$ and $\Omega(m)$. Our technique is the same as that of §2; namely, we reduce the original problem in $O(m)$ time to a collection of subproblems, each of which is small enough to solve by using a decision tree selected from a precomputed set of such trees.

Let $\{v, w\}$ be a nontree edge. Let $a(v, w) = \max\{c(x, y) \mid \{x, y\} \in T(v, w)\}$. Then T remains minimum until the edge cost of $\{v, w\}$ decreases by more than $c(v, w) - a(v, w)$. Similarly, let $\{v, w\}$ be a tree edge. Let $b(v, w) = \min\{c(x, y) \mid \{x, y\}$ is a nontree edge such that $\{v, w\} \in T(x, y)\}$. Then T remains minimum until the edge cost of $\{v, w\}$ increases by more than $b(v, w) - c(v, w)$. (See [18].)

The value of $a(v, w)$ for every nontree edge $\{v, w\}$ can be computed in $O(m)$ time by a simple extension of the verification algorithm in §2: instead of verifying that $c(v, w) \geq a(v, w)$, we compute $a(v, w)$ explicitly. Explicit computation of the $a(v, w)$ values can be done by computing, for each of the six replacement edges for an original edge $\{v, w\}$, the maximum-cost tree edge on the path in T joining the endpoints of the edge. These maxima can be computed by slightly modifying the algorithm of §2; the algorithm of Komlós actually computes the maxima along tree paths needed to handle the replacement edges within the microtrees.

Computing $b(v, w)$ for every tree edge $\{v, w\}$ is harder. We first replace the nontree edges exactly as in §2: each nontree edge $\{x, y\}$ is replaced by a set of up to six nontree edges, each of cost equal to $\{x, y\}$, in a way that preserves $b(v, w)$ for every tree edge $\{v, w\}$. In the process of performing this replacement, we choose a root r of T and compute subtree roots, subtrees, and microtrees exactly as in §2. After the replacement, each nontree edge $\{x, y\}$ is such that x and y are related in T, say, x is an ancestor of y. In addition, such an edge is of exactly one of three types:

Type 1: x is a subtree root, y is not a subtree root, and $x = p'(y)$, where p' is defined as in §2: $p'(y)$ is the deepest ancestor of y that is a subtree root;

Type 2: x and y are subtree roots;

Type 3: x and y are in the same microtree.

We compute each value $b(v, w)$ using the equation

$$b(v,w) = \min\{b_1(v,w),\, b_2(v,w),\, b_3(v,w)\}\,,$$

where b_i for $i = 1, 2, 3$ is defined exactly like $b(v, w)$, but with the minimum taken only over nontree edges of type i.

To compute the b_1-values, we begin by computing, for each vertex $v \neq r$, the value $\min_1(v) = \min\{c(x,y) \mid \{x,y\}$ is a type-1 edge such that $x = p'(y)$ is a proper ancestor of v and v is an ancestor of $y\}$. The $\min_1$-values can be computed in $O(m)$ time by visiting the vertices of T in postorder and applying the recurrence

$$\min_1(v) = \begin{cases} \infty & \text{if } v \text{ is a subtree root;} \\ \min\left(\{c(x,v) \mid \{x,v\} \text{ is a type-1 edge}\}\right. & \text{if } v \text{ is not a subtree root.} \\ \qquad \left.\cup\, \{\min_1(w) \mid p(w) = v\}\right) & \end{cases}$$

Then, for every vertex $v \neq r$, $b_1(p(v), v) = \min_1(v)$.

We compute the b_2-values in three steps. First, we form the graph G' as in §2 by adding to the macrotree T' each type-2 edge. Second, we compute, for each tree edge $\{v, w\}$ of the macrotree T', the value $b'(v,w) = \min\{c(x,y) \mid \{x,y\}$ is a type-2 edge such that $(v,w) \in T'(x,y)\}$. All the b'-values can be computed in $O(m\alpha(m,n'))$ time by applying the sensitivity analysis algorithm of Tarjan [18] to the graph G' and the tree T'. Choosing g (the size parameter for the macrotrees) to be $\Omega(\log^{(i)} n)$ for any fixed positive integer i results in an $O(m)$ time bound for this computation. Third, we compute, for each vertex $v \neq r$ in T, the value $\min_2(v) = \min\{b'(p'(y), y) \mid \{p'(y), y\}$ is an edge of T' such that y is a descendant of v and $p'(y)$ is a proper ancestor of $v\}$. The $\min_2$-values can be computed in $O(n)$ time by visiting the vertices of T in postorder and applying the recurrence

$$\min_2(v) = \begin{cases} b'(p'(v), v) & \text{if } v \text{ is a subtree root and} \\ \min\{\min_2(w) \mid p(w) = v\} & \text{if } v \text{ is not a subtree root.} \end{cases}$$

Then, for every vertex $v \neq r$, $b_2(p(v), v) = \min_2(v)$.

All that remains is to compute the b_3-values. Since the definition of a type-3 edge $\{x, y\}$ implies that x and y are in the same microtree, we can compute the b_3-values by adding the type-3 edges to the appropriate microtrees to form graphs $G_1, G_2, \ldots, G_k$ as in §2 and then process each G_i, T_i pair separately. We again use preprocessing to construct a fast decision-tree algorithm for each possible graph-tree pair and then use table look-up to select the correct algorithm for each actual pair G_i, T_i.

It is only in the construction of the decision trees that the randomized and deterministic algorithms differ. We first consider the deterministic case. A decision tree for the sensitivity analysis problem consists of a binary tree, each internal node of which specifies a comparison between the costs of two edges, and each leaf x of which provides a mapping f_x from the tree edges of the problem graph to the nontree edges, such

that the b_3-value of any edge e is $c(f_x(e))$, assuming that the edge costs are consistent with the outcome of the comparisons leading to leaf x. The algorithm of Tarjan [18] implies the existence of an $O(m\alpha(m,n))$-depth decision tree for the sensitivity analysis of an n-vertex, m-edge graph, and given spanning tree. Since $\alpha(g^2, g) = O(1)$ [19], these decision trees have depth $O(g^2)$.

For each possible connected graph with no more than g vertices and specified spanning tree, we construct a minimum-depth decision tree for sensitivity analysis by brute-force enumeration. We restrict our attention to complete binary trees, enumerating all possible decision trees of each possible depth in increasing order by depth until finding a correct one. A complete binary decision tree of depth d has $2^d - 1$ internal nodes and 2^d leaves. Each internal node corresponds to one of the less than g^4 possible binary comparisons of edge costs; each leaf can correspond to one of the less than g^{2g} possible mappings of the tree edges to the nontree edges. Thus there are less than $(g^4)^{2^d-1}(g^{2g})^{2^d} < g^{g2^{d+2}}$ possible decision trees of depth d, assuming $g \geq 2$. The total number of trees that must be considered before encountering a correct one is $O(g^{g2^{c_3 g^2}})$ for some suitably large constant c_3. This is $O(2^{2^{g^3}})$. The space needed to store a decision tree of depth d is $O(2^d g \log g) = O(2^{g^3})$, if $d = O(g^2)$. To determine whether a particular decision tree correctly solves the sensitivity analysis problem, it suffices to test that the correct answer is obtained for each of the at most $(g^2)! = O(2^{g^3})$ possible permutations of edge costs. Testing one permutation requires $O(g^2)$ time. The time to test a particular decision tree is thus $O(g^2 2^{g^3}) = O(2^{g^4})$, and the time to find a minimum-depth decision tree is $O(2^{g^4} 2^{2^{g^3}}) = O(2^{2^{g^4}})$. If we choose $g = c_4(\log\log n)^{1/4}$ for some sufficiently small constant c_4, then the time to find minimum-depth decision trees for all possible graph-tree pairs is $O(n)$, as is the space needed to store them in a table.

We compute the b_3-values for all tree edges in a microtree T_i by indexing the look-up table with the code string of the pair G_i, T_i to get a decision tree and evaluating the decision tree with the given edge costs.

The total time needed for sensitivity analysis is $O(m)$ plus time proportional to the sum of the minimum numbers of comparisons needed to perform sensitivity analysis for all of the G_i, T_i pairs. Performing sensitivity analysis for all of the G_i, T_i pairs is at most a constant factor more time-consuming than performing sensitivity analysis for a worst-case n-vertex, m-edge graph. Thus the sensitivity analysis algorithm runs in time minimum to within a constant factor, assuming that only binary comparisons between edge costs are used as tests.

In the randomized case, we replace the deterministic decision trees used for sensitivity analysis of the microtrees by randomized decision trees. In a randomized decision tree, each internal node corresponds either to a comparison of two edge costs or to a test of a distinct random bit. As in the deterministic case, we require every path of the decision tree to give the correct answer, but as a measure of the complexity of the tree we use the weighted average depth of a leaf, rather than the worst-case depth, taking the weight of a leaf to be $1/2^i$, where i is the number of tests of random bits along the path from the root to the leaf.

Goddard, King, and Shulman [9] have found a randomized algorithm to compute the maxima of n subsets of an ordered universe of size n in $O(n)$ comparisons on the average. Their result, together with the observation of King [13] that their algorithm needs only $O(n)$ random bits on the average, implies the existence of a randomized decision tree of average depth $O(m)$ for the sensitivity analysis problem. Such a decision tree can be converted into a decision tree of $O(m)$ average depth and $O(m \log m)$ worst-case

depth by trimming the decision tree at depth $m \log m$ and replacing each subtree that was cut out by a decision tree that merely sorts by cost the edges of the problem graph. A brute-force enumeration can be used to find minimum-average-depth randomized decision trees for all possible microtree problems. The details mimic the deterministic case, so we omit them. The Goddard–King–Shulman result then implies that the resulting randomized sensitivity analysis algorithm runs in $O(m)$ expected time for a suitable choice of the microtree size bound g.

4. Concluding remarks. We have illustrated by means of two related examples a general technique of speeding up divide-and-conquer algorithms by a preprocessing and table look-up technique. A curious phenomenon is that the technique can give algorithms running in time minimum to within a constant factor, but for which we cannot presently offer a tight asymptotic time analysis. This is the case for our deterministic minimum spanning tree sensitivity analysis algorithm and for Larmore's convex matrix searching algorithm [15]; both have running times somewhere between linear and an inverse Ackerman function times linear. Providing tight analysis of these algorithms amounts to bounding the number of comparisons needed to solve the corresponding problems. Obtaining tight bounds remains open. A related question is whether the randomized maxima-finding algorithm of Goddard, King, and Shulman can be made deterministic. Another question is whether our model for computations not involving edge costs can be weakened from a random-access machine to a pointer machine. The only feature of a random-access machine we have actually used is the ability to make multiway branches to look up the special-purpose algorithms for processing the microtrees. It may well be the case that binary branching suffices for this purpose, and hence that our results hold for pointer machines with binary comparison of edge costs (and edge cost subtraction for sensitivity analysis).

The technique we have illustrated is not limited to comparison-based problems. We can allow arithmetic operations in the decision trees to be used to solve the subproblems. Testing the correctness of such a decision tree amounts to testing the validity of a first-order sentence about the real numbers. Such sentences can be tested in double-exponential time [1], which suffices for the use of the method: we merely reduce the size of the subproblems to double-logarithmic, triple-logarithmic, or further, as needed. As an example, the technique can be applied to the $O(n \log^* n)$-time algorithm of Chazelle [3] for triangulating a simple n-sided polygon, to produce an algorithm running in time minimum to within a constant factor. The bound for this algorithm is in fact $O(n)$ because of the even more recent result of Chazelle [4], giving an explicitly linear-time algorithm. Further applications remain to be discovered.

REFERENCES

[1] M. BEN-OR, D. KOZEN, AND J. REIF, *The complexity of elementary algebra and geometry*, J. Comput. System Sci., 32 (1986), pp. 251–264.

[2] H. BOOTH AND J. WESTBROOK, *Linear algorithms for analysis of minimum spanning and shortest path trees in planar graphs*, Yale Univ., Dept. of Computer Science, TR-768, Feb. 1990; also Algorithmica, to appear.

[3] B. CHAZELLE, *Efficient polygon triangulation*, CS-TR-249-90, Princeton Univ., Princeton, NJ, Feb. 1990.

[4] ———, *Triangulating a simple polygon in linear time*, in Proc. 31st Annual IEEE Symposium on Foundations in Comput. Sci., Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 220–230; also Discrete Comput. Geom., 6 (1991), pp. 485–524.

[5] M.L. FREDMAN, *New bounds on the complexity of the shortest path problem*, SIAM J. Comput., 5 (1976), pp. 83–89.

[6] M.L. FREDMAN AND D.E. WILLARD, *Trans-dichotomous algorithms for minimum spanning trees and shortest paths*, Proc. 31st Annual IEEE Symposium on Foundations in Comput. Sci., Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990, pp. 719–725.

[7] H.N. GABOW, Z. GALIL, T. SPENCER, AND R.E. TARJAN, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica, 6 (1986), pp. 109–122.

[8] H.N. GABOW AND R.E. TARJAN, *A linear-time algorithm for a special case of disjoint set union*, J. Comput. System Sci., 30 (1985), pp. 209–221.

[9] W. GODDARD, V. KING, AND L. SCHULMAN, *Optimal randomized algorithms for local sorting and set-maxima*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, ACM Press, New York, New York, 1990, pp. 45–53.

[10] R.L. GRAHAM AND P. HELL, *On the history of the minimum spanning tree problem*, Ann. Hist. Comput., 7 (1985), pp. 43–47.

[11] D. HAREL, *A linear time algorithm for finding dominators in flow graphs and related problems*, in Proc. 17th Annual ACM Symposium on Theory of Computing, Salem, MA, 1985, pp. 185–194.

[12] D. HAREL AND R.E. TARJAN, *Fast algorithms for finding nearest common ancestors*, SIAM J. Comput., 13 (1984), pp. 338–355.

[13] V. KING, personal communication.

[14] J. KOMLÓS, *Linear verification for spanning trees*, Combinatorica, 5 (1985), pp. 57–65.

[15] L.L. LARMORE, *An optimal algorithm with unknown time complexity for convex matrix searching*, Inform. Process. Lett., 36 (1990), pp. 147–151.

[16] B. SCHIEBER AND U. VISHKIN, *On finding lowest common ancestors: simplification and parallelization*, SIAM J. Comput., 17 (1988), pp. 1253–1262.

[17] R.E. TARJAN, *Applications of path compressions on balanced trees*, J. Assoc. Comput. Mach., 26 (1979), pp. 690–715.

[18] ———, *Sensitivity analysis of minimum spanning trees and shortest path trees*, Inform. Process. Lett., 14 (1982), pp. 30–33; Corrigendum, Ibid. 23 (1986), p. 219.

[19] ———, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

SIAM J. COMPUT.
Vol. 21, No. 6, pp. 1193–1198, December 1992

© 1992 Society for Industrial and Applied Mathematics
013

OPTIMAL ALGORITHMS FOR MULTIPLICATION IN CERTAIN FINITE FIELDS USING ELLIPTIC CURVES*

MOHAMMAD AMIN SHOKROLLAHI†

Abstract. Using the results given in [D. V. Chudnovsky and G. V. Chudnovsky, *Proc. Nat. Acad. Sci. USA*, 84 (1987), pp. 1739–1743] and [W. C. Waterhouse, *Ann. Sci. École Norm. Sup.*, 4 (1969), pp. 521–560], it is proven that the rank (=bilinear complexity of multiplication) of the finite field $\mathbb{F}_{q^n}$ viewed as an $\mathbb{F}_q$-algebra is $2n$ if n satisfies $\frac{1}{2}q+1 < n < \frac{1}{2}(q+1+\epsilon(q))$. Here $\epsilon(q)$ is the greatest integer $\leq 2\sqrt{q}$, which is prime to q if q is not a perfect square and $\epsilon(q) = 2\sqrt{q}$ if q is a perfect square.

Key words. elliptic curves, bilinear complexity, finite fields

AMS(MOS) subject classifications. 14H52, 03D15, 11Y16

1. Introduction. Let K be a field and L a simple finite extension field of K. The *rank* $R(L/K)$ of L over K is defined to be the bilinear complexity of multiplication in L/K, where K is regarded as the field of scalars [7]. Denoting by L^* the dual of L as a vector space over K we thus have

$$R(L/K) = \min\Big\{ r \in \mathbb{N} \;\Big|\; \exists\, u_i, v_i \in L^*, w_i \in L\; \forall\, a, b \in L : \quad ab = \sum_{i=1}^{r} u_i(a)v_i(b)w_i \Big\}.$$

Let n denote the degree $(L : K)$. It is known that $R(L/K) \geq 2n-1$ with equality holding if and only if $|K| \geq 2n-2$ [6]. The optimal multiplication algorithms realizing the lower bound $2n-1$ all belong to the class of *interpolation algorithms* [12]. These algorithms are based on the principle of reconstruction of polynomial products by Lagrange-interpolation [3]. In other words, these algorithms can be viewed as interpolation algorithms on the projective line over $\mathbb{F}_q$.

Noticing that a switch to arbitrary algebraic curves results in the existence of more rational points than in the case of a projective line, D. V. Chudnovsky and G. V. Chudnovsky generalized the existing algorithms to projective curves having additional arithmetic properties [3]. Using special, well-studied algebraic curves, they were the first to show that for an infinite set of prime powers q, the rank of $\mathbb{F}_{q^n}/\mathbb{F}_q$ is asymptotically bounded from above by a linear function of n.

Our paper is concerned with the application of the algorithm invented by Chudnovsky and Chudnovsky to the first nontrivial class of algebraic curves, the so-called *elliptic curves*. Elliptic curves play an important role in different areas of mathematics and computer science and have been studied very well during the last decades.

Besides classical results about elliptic curves, we are interested in the question of the existence of elliptic curves over the finite field $\mathbb{F}_q$ with as many rational points as possible. This question has been answered by Waterhouse [11]. Combining the results of Waterhouse with the mentioned interpolation algorithm on algebraic curves we are able to compute the exact rank of certain field extensions of the finite field $\mathbb{F}_q$ (Theorem 5).

The question of generating the optimal algorithms, the existence of which is proved in this paper, calls for other types of methods. We will discuss this problem in detail in [9].

*Received by the editors September 6, 1990; accepted for publication (in revised form) November 1, 1991.
†Institut für Informatik V, Universität Bonn, Romerstraße 164, 5300 Bonn, Germany.

The paper is organized as follows. In §2 we summarize some well-known results about elliptic function fields (which are the function fields of the elliptic curves). Section 3 is devoted to the application of a slight modification of the algorithm presented in [3] to elliptic function fields. Section 4 discusses the question of the existence of elliptic function fields, having those additional arithmetic properties required by the algorithm of §3. Section 5 is devoted to the formulation and proof of the main theorem.

2. Basic facts about elliptic function fields. In this section we are going to introduce some basic notations and summarize well-known results about elliptic function fields over finite fields. All these facts can be found in standard textbooks such as [1], [2], [5], [10].

Let $K/\mathbb{F}_q$ be an elliptic function field with constant field $\mathbb{F}_q$, i.e., $\mathbb{F}_q$ is assumed to be algebraically closed in K. By $\mathbb{P}(K/\mathbb{F}_q)$ we denote the set of prime divisors of $K/\mathbb{F}_q$. $\mathbb{D}(K/\mathbb{F}_q)$ denotes the group of divisors of $K/\mathbb{F}_q$ defined to be the free abelian group over $\mathbb{P}(K/\mathbb{F}_q)$. The relation $\leq$ defined by

$$\sum_{\mathfrak{p}} a_{\mathfrak{p}}\mathfrak{p} \leq \sum_{\mathfrak{p}} b_{\mathfrak{p}}\mathfrak{p} \quad :\Longleftrightarrow \quad \forall \mathfrak{p} \ \ a_{\mathfrak{p}} \leq b_{\mathfrak{p}}$$

is a partial order on $\mathbb{D}(K/\mathbb{F}_q)$.

For $\mathfrak{p} \in \mathbb{P}(K/\mathbb{F}_q)$ we define $K\mathfrak{p}$ to be the residue class field of $\mathfrak{p}$. It is well known that $K\mathfrak{p}$ is a finite extension of $\mathbb{F}_q$. The index $(K\mathfrak{p} : \mathbb{F}_q)$ is denoted by $\deg(\mathfrak{p})$ and is called the *degree of* $\mathfrak{p}$. The map $\mathfrak{p} \mapsto \deg(\mathfrak{p})$ can be uniquely extended to $\mathbb{D}(K/\mathbb{F}_q)$ by

$$\deg\left(\sum_{\mathfrak{p}} a_{\mathfrak{p}}\mathfrak{p}\right) := \sum_{\mathfrak{p}} a_{\mathfrak{p}} \deg(\mathfrak{p}).$$

To every nonvanishing function f in K we can associate the divisor

$$(f) := \sum_{\mathfrak{p}} \operatorname{ord}_{\mathfrak{p}}(f)\, \mathfrak{p},$$

called the *principal divisor of* f. Here $\operatorname{ord}_{\mathfrak{p}}(f)$ denotes the $\mathfrak{p}$-order of f. The principal divisors form a subgroup $\mathbb{H}(K/\mathbb{F}_q)$ of $\mathbb{D}(K/\mathbb{F}_q)$, isomorphic to $K^{\times}/\mathbb{F}_q^{\times}$. All principal divisors are of degree zero. For a divisor $\mathfrak{A} \in \mathbb{D}(K/\mathbb{F}_q)$, the set $\mathfrak{A} + \mathbb{H}(K/\mathbb{F}_q)$ is called the *class of* $\mathfrak{A}$. Since principal divisors are of degree zero, the degree map is constant on classes of divisors.

For $\mathfrak{A} \in \mathbb{D}(K/\mathbb{F}_q)$ we denote by $\mathcal{L}(\mathfrak{A})$ the *linear space attached to* $\mathfrak{A}$, which besides zero contains all nonvanishing functions f of K with $(f) \geq -\mathfrak{A}$. $\mathcal{L}(\mathfrak{A})$ is even a vector space of finite dimension $\dim(\mathfrak{A})$ over $\mathbb{F}_q$. The number $\dim(\mathfrak{A})$ is called the *dimension* of the divisor $\mathfrak{A}$. Like the degree, the dimension is also a class function.

The theorem of Riemann–Roch [10, Thm. 5.4] relates the dimension and the degree of an arbitrary divisor of $K/\mathbb{F}_q$.

THEOREM 1 (Theorem of Riemann–Roch). *Let $K/\mathbb{F}_q$ be an elliptic function field and $\mathfrak{A} \in \mathbb{D}(K/\mathbb{F}_q)$ be an arbitrary divisor. Then we have*

$$\dim(\mathfrak{A}) = \begin{cases} 0 & \text{if } \deg(\mathfrak{A}) < 0, \\ 1 & \text{if } \mathfrak{A} \in \mathbb{H}(K/\mathbb{F}_q), \\ \deg(\mathfrak{A}) & \text{otherwise.} \end{cases}$$

The set of prime divisors of degree one of $K/\mathbb{F}_q$ is denoted by $\mathbb{P}_1(K/\mathbb{F}_q)$. Since $K/\mathbb{F}_q$ is elliptic, this set is not empty. Hasse [8] proved the inequality

$$q+1-2\sqrt{q} \le |\mathbb{P}_1(K/\mathbb{F}_q)| \le q+1+2\sqrt{q}. \tag{1}$$

We shall be interested in elliptic function fields with as many prime divisors of degree one as possible. For example, if q is a perfect square, we ask if there are elliptic function fields having $q+1+2\sqrt{q}$ prime divisors of degree one.

An answer to this question can be given using a more general theorem of Waterhouse [11], which gives necessary and sufficient conditions for the existence of an elliptic function field having $t+q+1$ prime divisors of degree one if t is a given natural number satisfying $|t| \le 2\sqrt{q}$ in view of (1). Before stating the result, let us introduce the function ϵ defined by

$$\epsilon(q) := \begin{cases} \text{greatest integer} \le 2\sqrt{q} \text{ prime to } q & \text{if } q \text{ is not a perfect square,} \\ 2\sqrt{q} & \text{if } q \text{ is a perfect square.} \end{cases}$$

THEOREM 2 (Waterhouse [11]). *Let q be a prime power. Then there exists an elliptic function field over $\mathbb{F}_q$ having $q+1+\epsilon(q)$ prime divisors of degree one.*

3. Interpolation in elliptic function fields. In this section we shall discuss a modified version of a bilinear algorithm due to Chudnovsky and Chudnovsky [3] for multiplication in a finite extension of $\mathbb{F}_q$. We shall first state the result.

THEOREM 3. *Let q be a prime power and n be a natural number. Suppose that there exists an elliptic function field $K/\mathbb{F}_q$ satisfying the following conditions:*

(1) *K contains a prime divisor $\mathfrak{p}$ of degree n;*

(2) *K contains a divisor $\mathfrak{D}$ of degree n, the class of which is different from that of $\mathfrak{p}$ and for which $\mathrm{ord}_{\mathfrak{P}}(\mathfrak{D}) = 0$ for all prime divisors $\mathfrak{P}$ of degree one of $K/\mathbb{F}_q$;*

(3) $|\mathbb{P}_1(K/\mathbb{F}_q)| > 2n$.

Then we have

$$R(\mathbb{F}_{q^n}/\mathbb{F}_q) \le 2n.$$

Proof. The proof uses the same method as in [3]. Let $\mathfrak{p}$ and $\mathfrak{D}$ be as in the assumptions of the theorem. Since $\mathfrak{p}$ is of degree n, the residue class field $K\mathfrak{p}$ of $\mathfrak{p}$ is isomorphic to $\mathbb{F}_{q^n}$. Further, since $\mathfrak{D}$ and $\mathfrak{p}$ belong to different classes and $\mathfrak{p}$ is assumed to be prime, we have $\mathrm{ord}_{\mathfrak{p}}(\mathfrak{D}) = 0$ showing that $\mathcal{L}(\mathfrak{D})$ is contained in the valuation ring of $\mathfrak{p}$. Let $\kappa : \mathcal{L}(\mathfrak{D}) \to K\mathfrak{p}$ denote the restriction of the residue class mapping on $\mathcal{L}(\mathfrak{D})$. Thus κ defines a vector space homomorphism. The kernel of κ is $\mathcal{L}(\mathfrak{D}-\mathfrak{p})$. Since $\mathfrak{D}$ and $\mathfrak{p}$ belong to different classes, $\mathfrak{D}-\mathfrak{p}$ is not principal. Theorem 1 implies now that $\mathcal{L}(\mathfrak{D}-\mathfrak{p})$ is trivial, which shows that κ is injective. By Theorem 1 $\dim(\mathfrak{D}) = n$; hence κ is an isomorphism. So there exists a basis $f_1, \cdots, f_n$ of $\mathcal{L}(\mathfrak{D})$, which is mapped by κ onto a basis of $K\mathfrak{p}$ over $\mathbb{F}_q$.

Let $\{g_1, \cdots, g_{2n}\}$ be a basis of $\mathcal{L}(2\mathfrak{D})$. Then there exist elements $B_{ij}^r \in \mathbb{F}_q$ such that

$$f_i f_j = \sum_{r=1}^{2n} B_{ij}^r g_r.$$

Furthermore, $\mathcal{L}(2\mathfrak{D})$ is also contained in the valuation ring of $\mathfrak{p}$. By abuse of notation, let us denote the extension of κ to $\mathcal{L}(2\mathfrak{D})$ again by κ. Then there exist $c_r^m \in \mathbb{F}_q$ such that

we have

$$\kappa(g_r) = \sum_{m=1}^{n} c_r^m \kappa(f_m), \qquad r = 1, \cdots, 2n.$$

Let $\sum_{i=1}^n x_i\kappa(f_i)$ and $\sum_{j=1}^n y_j\kappa(f_j)$ be two arbitrary elements of $K\mathfrak{p} = \mathbb{F}_{q^n}$. Then we get

$$\left(\sum_{i=1}^{n} x_i\kappa(f_i)\right)\left(\sum_{j=1}^{n} y_j\kappa(f_j)\right) = \sum_{m=1}^{n}\left(\sum_{r=1}^{2n}\left(\sum_{i,j=1}^{n} x_i y_j B_{ij}^r\right) c_r^m\right)\kappa(f_m). \tag{2}$$

Let the bilinear forms $Z_1, \cdots, Z_{2n}$ be defined by

$$Z_r := \sum_{i,j=1}^{n} x_i y_j B_{ij}^r, \qquad r = 1, \cdots, 2n.$$

By (2) every bilinear algorithm of length l for computing $Z_1, \cdots, Z_{2n}$ produces an algorithm of length l for multiplication in $\mathbb{F}_{q^n}$ over $\mathbb{F}_q$. Hence we have

$$R(\mathbb{F}_{q^n}/\mathbb{F}_q) \leq R(\{Z_1, \cdots, Z_{2n}\}),$$

where $R(\{Z_1, \cdots, Z_{2n}\})$ denotes the bilinear complexity of the set of bilinear forms $\{Z_1, \cdots, Z_{2n}\}$ (certificate [6, Chap. I]).

For the computation of $Z_1, \cdots, Z_{2n}$ we are going to use the interpolation algorithm presented in [3].

Let $\{\mathfrak{P}_1, \cdots, \mathfrak{P}_N\}$ be the set of prime divisors of degree one of $K/\mathbb{F}_q$. Further, let the matrix Γ be defined by

$$\Gamma := \begin{pmatrix} g_1(\mathfrak{P}_1) & \cdots & g_{2n}(\mathfrak{P}_1) \\ \vdots & \ddots & \vdots \\ g_1(\mathfrak{P}_N) & \cdots & g_{2n}(\mathfrak{P}_N) \end{pmatrix}.$$

This matrix is defined since by the assumptions of the theorem $\mathrm{ord}_{\mathfrak{P}}(\mathfrak{D}) = 0$ for all $\mathfrak{P} \in \mathbb{P}_1(K/\mathbb{F}_q)$. The rank of Γ equals $2n$: Consider the homomorphism

$$\begin{aligned} \gamma: \quad \mathcal{L}(2\mathfrak{D}) &\to \mathbb{F}_q^N \\ g &\mapsto (g(\mathfrak{P}_1), \cdots, g(\mathfrak{P}_N)). \end{aligned}$$

Γ is the representation matrix of γ with respect to the basis $\{g_1, \cdots, g_{2n}\}$ in $\mathcal{L}(2\mathfrak{D})$ and the canonical basis in $\mathbb{F}_q^N$. The kernel of γ is $\mathcal{L}(2\mathfrak{D} - (\mathfrak{P}_1 + \cdots + \mathfrak{P}_N))$, which is trivial by Theorem 1 since N is assumed to be larger than $2n$. So γ is injective, meaning that Γ has rank $2n$. Without loss of generality, suppose that the first $2n$ rows of Γ are linearly independent. Denote by Γ_0 the matrix formed by these rows of Γ. We define further

$$X_s := \sum_{i=1}^{n} x_i f_i(\mathfrak{P}_s), \qquad Y_s := \sum_{j=1}^{n} y_j f_j(\mathfrak{P}_s), \quad s = 1, \cdots, 2n.$$

Now we compute $(X_1Y_1, \cdots, X_{2n}Y_{2n})$. This step of the algorithm requires $2n$ essential multiplications.

Since by (2) we have

$$\Gamma_0(Z_1,\cdots,Z_{2n})^\top = (X_1Y_1,\cdots,X_{2n}Y_{2n})^\top,$$

we get the desired bilinear forms $Z_1,\cdots,Z_{2n}$ without further essential multiplications. This proves the theorem. □

Note that in the algorithm presented above $\mathfrak{D}$ does not need to be integral, a condition assumed in [3].

4. Technical tools. This section is rather technical and is primarily concerned with the question of which elliptic function fields satisfy the conditions of Theorem 3.

The first problem we are concerned with is that of the existence of prime divisors of given degree in an elliptic function field. The next lemma answers this question.

LEMMA 1. *Let q be a prime power, $q \geq 4$. Further, let n be a natural number satisfying $n > \frac{1}{2}q + 1$. Then every elliptic function field over $\mathbb{F}_q$ contains a prime divisor of degree n.*

Proof. In [3] it is proved that the number N_n of prime divisors of degree n of an algebraic function field of genus g over $\mathbb{F}_q$ satisfies the inequality

$$N_n \geq \frac{1}{n}\left(q^n - q^{n/2}(4g+q)\right).$$

Since the genus of an elliptic function field is one, it suffices to prove the inequality

$$q^{n/2} > 4 + q.$$

Because $q \geq 4$ is assumed, the stronger inequality $q^{n/2} > 2q$ also yields the result. But this inequality is satisfied for all n with $n > 2\log 2/\log q + 2$. Now the assertion follows since for $q \geq 4$ we have $\frac{1}{2}q + 1 \geq 2\log 2/\log q + 2$. □

For proving the existence of the divisor $\mathfrak{D}$ we first have to show that there exist two different divisor classes of degree n. This is the content of the next lemma.

LEMMA 2. *Let $K/\mathbb{F}_q$ be an elliptic function field that has at least two different prime divisors of degree one. Then K contains two different divisor classes of degree n for every natural number n.*

Proof. Let $\mathfrak{o}$ be a divisor of degree one of $K/\mathbb{F}_q$. Further, let $\mathfrak{p}_1$ and $\mathfrak{p}_2$ be two different prime divisors of degree one of $K/\mathbb{F}_q$ and $\mathfrak{A}$ be a divisor of degree n of $K/\mathbb{F}_q$. Then $\mathfrak{A}+\mathfrak{p}_1-\mathfrak{o}$ and $\mathfrak{A}+\mathfrak{p}_2-\mathfrak{o}$ clearly belong to two different classes of degree n. □

With the foregoing two lemmas at hand we can prove the existence of the divisor $\mathfrak{D}$ of Theorem 3.

THEOREM 4. *Let q be a prime power, $q \geq 4$, and n be an integer with $n > \frac{1}{2}q + 1$. Further, let $K/\mathbb{F}_q$ be an elliptic function field with at least two prime divisors of degree one. Then K contains a prime divisor $\mathfrak{p}$ of degree n and a divisor $\mathfrak{D}$ of degree n not belonging to the class of $\mathfrak{p}$ such that for all $\mathfrak{P} \in \mathbb{P}_1(K/\mathbb{F}_q)$ we have $\mathrm{ord}_{\mathfrak{P}}(\mathfrak{D}) = 0$.*

Proof. The existence of $\mathfrak{p}$ follows from Lemma 1. Let C_1 denote the class of $\mathfrak{p}$. By Lemma 2 K contains a class C_2 of divisors of degree n with $C_1 \neq C_2$. By [5, Lem. 1, p. 71] C_2 contains a divisor $\mathfrak{D}$ such that $\mathrm{ord}_{\mathfrak{P}}(\mathfrak{D}) = 0$ for all $\mathfrak{P} \in \mathbb{P}_1(K/\mathbb{F}_q)$. This proves the theorem. □

5. The main result. This section is devoted to the formulation and proof of the main theorem of this paper.

THEOREM 5. *Let q be a prime power and n an integer satisfying $\frac{1}{2}q + 1 < n < \frac{1}{2}(q + 1 + \epsilon(q))$. Then we have*

$$R(\mathbb{F}_{q^n}/\mathbb{F}_q) = 2n.$$

Proof. Since for $q \in \{2,3\}$ the assertion of the theorem is empty, let us suppose that $q \geq 4$. Let $K/\mathbb{F}_q$ be an elliptic function field with $q+1+\epsilon(q)$ prime divisors of degree one. (The existence of K follows from Theorem 2.) Since $q \geq 4$, the function field $K/\mathbb{F}_q$ contains more than two prime divisors of degree one. Applying Lemmas 1 and 2, we get the existence of a prime divisor $\mathfrak{p}$ of degree n as well as the existence of the divisor $\mathfrak{D}$ with the conditions stated in Theorem 3. Further, the assumption on n implies $2n < q+1+\epsilon(q)$. Hence, Theorem 3 yields the assertion $R(\mathbb{F}_{q^n}/\mathbb{F}_q) \leq 2n$.

Now by [6, Thm. 1.4], if $q < 2n-2$ we have $R(\mathbb{F}_{q^n}/Fq) > 2n-1$. So the assertion of the theorem follows. □

As a corollary to the above theorem we get the following.

COROLLARY 1. *Let q be a prime power that is a perfect square. Further, let n be an integer satisfying $\frac{1}{2}q+1 < n < \frac{1}{2}(q+1+2\sqrt{q})$. Then we have*

$$R(\mathbb{F}_{q^n}/\mathbb{F}_q) = 2n.$$

Acknowledgment. The author wants to thank Michael Clausen for many helpful discussions during the research of this paper and two anonymous referees for essential suggestions and corrections of a former version of Theorem 3.

REFERENCES

[1] E. ARTIN, *Algebraic Numbers and Algebraic Functions*, Gordon and Breach, New York, 1977.

[2] C. CHEVALLEY, *Introduction to Algebraic Functions of One Variable*, American Mathematical Society, New York, 1958.

[3] D. V. CHUDNOVSKY AND G. V. CHUDNOVSKY, *Algebraic complexities and algebraic curves over finite fields*, Proc. Nat. Acad. Sci. USA, 84 (1987), pp. 1739–1743.

[4] ———, *Algebraic complexities and algebraic curves over finite fields*, Res. Report RC 12065, IBM Research Center, Yorktown Heights, 1987.

[5] M. DEURING, *Lectures on the theory of algebraic functions of one variable*, Lecture Notes in Math. 314, Springer-Verlag, Heidelberg, New York, Tokyo, 1973.

[6] H. F. DE GROOTE, *Characterization of division algebras of minimal rank and the structure of their algorithm varieties*, SIAM J. Comput., 12 (1983), pp. 101–117.

[7] ———, *Lectures on the complexity of bilinear problems*, Lecture Notes in Computer Science, 245, Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1985.

[8] H. HASSE, *Beweis des Analogons der Riemannschen Vermutung für die Artinschen und F. K. Schmidtschen Kongruenzzetafunktionen in gewissen elliptischen Fällen*, Nachr. der math. Gesellschaft zu Göttingen, 3 (1933), pp. 253–262.

[9] M. A. SHOKROLLAHI, *Efficient randomized generation of optimal algorithms for multiplication in certain finite fields*, Computational Complexity, to appear.

[10] J. H. SILVERMAN, *The Arithmetic of Elliptic Curves*, Springer-Verlag, New York, Berlin, Heidelberg, Tokyo, 1986.

[11] W. C. WATERHOUSE, *Abelian varieties over finite fields*, Ann. Sci. École Norm. Sup., 4 (1969), pp. 521–560.

[12] S. WINOGRAD, *On multiplication in algebraic extension fields*, Theoret. Comput. Sci., 8 (1979), pp. 359–377.